Оглавление

Введение

Такие устройства, как метроном, часто используются музыкантами для репетиций и тренировок. Причём нередко они используют в таких случаях именно программное обеспечение, моделирующее работу этого устройства. При этом некоторая реалистичность внешнего вида метронома при его программной реализации даёт ощущение использования реального метронома.

Работа любого метронома построена на колебательном движении. Перед запуском метронома указывается темп (единицы измерения – количество ударов в минуту или beats per minute, сокращённо bpm), под которым подразумевается частота колебаний маятника в метрономе, на основе которой в свою очередь и происходит работа этого метронома. При наибольших отклонениях маятника, равных амплитуде его колебаний, происходит очередной удар метронома.

Классический вариант метронома, который и должен быть представлен в этом программном продукте, состоит из корпуса, имеющего вид правильной прямоугольной пирамиды, и маятника, установленного на лицевой стороне этого корпуса.

Целью данной работы является разработка ПО, в котором реализованы работа метронома и возможность его масштабирования, переноса и поворота.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить колебательное движение, понятия и формулы, необходимые для его моделирования;
- проанализировать возможные алгоритмы удаления невидимых линий объекта (метронома);

- проанализировать возможные алгоритмы учёта освещённости объекта и падающих теней;
- изучить алгоритмы масштабирования, переноса и поворота трёхмерного изображения;
- спроектировать архитектуру программного обеспечения;
- реализовать выбранные алгоритмы;
- разработать описанное программное обеспечение
- протестировать разработанное программное обеспечение.

1 Аналитическая часть

В этом разделе будет проанализирована предметная область и рассмотрены алгоритмы моделирования колебательного движения и синтеза реалистического изображения.

1.1 Постановка задачи

Необходимо обеспечить отрисовку трёхмерной модели метронома с возможностью его запуска и остановки, учитывая освещение от находящихся на сцене источников света. Также необходимо обеспечить возможность масштабирования, перемещения и поворота модели и камеры.

1.2 Описание и формализация объектов сцены

Для описания трёхмерных геометрических объектов существуют 3 модели [4]: каркасная, поверхностная и объёмная. Для реализации поставленной задачи наиболее подходящей будет поверхностная модель, так как, по сравнению с каркасной, она может дать более реалистичное и понятное пользователю изображение, и в то же время для неё требуется меньше памяти, чем для объёмной модели.

В свою очередь поверхностная модель может задаваться параметрически или полигональной сеткой.

Так как в модели метронома отсутствуют поверхности вращения, то использовать параметрическое представление невыгодно. Следовательно, для представления модели будет использоваться полигональная сетка.

Для полигональной модели существуют несколько способов представления:

- вершинное представление (вершины хранят указатели на соседние вершины). В таком случае для отрисовки нужно будет обойти все данные по списку, что может занимать достаточно много времени при переборе;

- список граней (множество граней и вершин). Этот способ представления удобен при манипуляциях с данными;
- таблица углов (вершины хранятся в таблице, обход которой задают полигоны). Более компактное и производительное для нахождения полигонов, но операции по замене достаточно медлительны.

Наиболее подходящим представлением сцены в условиях данной задачи будет представление в виде списка граней.

В данной работе сцена состоит из следующих объектов:

- трёхмерная модель метронома, в свою очередь состоящая из двух подобъектов: корпуса и маятника. Эти объекты заданы файлами в формате obj;
- источники света, положение которых регулируется координатами (x, y, z).

1.3 Колебательное движение

1.3.1 Физические основы

Колебательные движения в метрономе можно считать колебаниями математического маятника, так как маятник в метрономе представляет собой тело небольших размеров, расположенное на стержне, масса которого пренебрежимо мала по сравнению с массой тела [2].

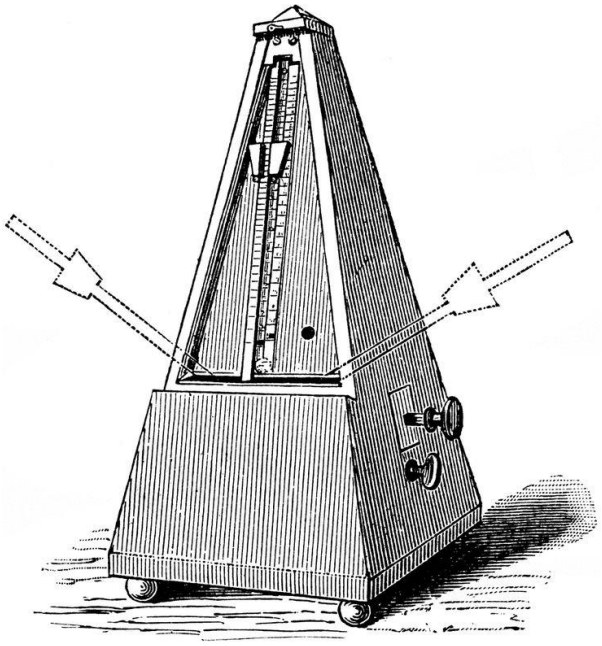

Рис. 1.1: Внешний вид классического метронома и его маятника

В дальнейшем под словом маятник будет подразумеваться именно математический маятник.

Понятия, связанные с колебательным движением, использующиеся в данной работе:

- амплитуда колебаний математического маятника (A) – наибольшее отклонение этого маятника от положения равновесия;
- частота колебаний (ν) – величина, показывающая, какое число полных колебаний совершает тело за единицу времени;
- циклическая частота колебаний (ω_0) – число полных колебаний за время, равное 2π;
- период колебаний (T) – время, за которое телом совершается одно полное колебание;
- фаза колебаний – состояние колебательной системы в данный момент времени.

При колебаниях в одной плоскости маятник движется по дуге окружности радиуса L, где L – длина стержня маятника.

Колебания маятника описываются уравнением:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin\theta = 0, \quad (1.1)$$

где неизвестная функция $\theta(t)$ – это угол отклонения маятника от положения равновесия в момент времени t, выраженный в радианах [3].

При малых колебаниях (амплитуда до 15-20 градусов) $\sin\theta$ можно заменить на θ.

Решением уравнения (1.1) является функция:

$$\theta(t) = A\cos(\omega_0 t + \alpha), \quad (1.2)$$

где α – начальная фаза колебаний (при t = 0) [1].

Связь между периодом и частотой колебаний:

$$T = \frac{1}{\nu} \quad (1.3)$$

$$\omega_0 = \frac{2\pi}{T} = 2\pi\nu \quad (1.4)$$

Взяв первую производную от выражения (1.2) и приравняв её к нулю, можно найти моменты времени t, при которых тело будет находиться в положении, соответствующем амплитуде его колебаний. Получается:

$$t_A = \frac{\frac{\pi}{2} + \pi k - \alpha}{\omega_0} \quad (1.5)$$

Где k = 0, 1, 2, …

Для метронома $\alpha = 0$.

При этом для малых (гармонических) колебаний период не зависит от амплитуды.

1.3.2 Моделирование колебаний

В компьютерной графике моделировать колебания можно несколькими способами.

Первый способ основан на физических законах и уравнениях. Он состоит в том, чтобы использовать выражение (1.2) для расчёта угла отклонения

маятника от положения равновесия в каждый момент времени. Исходя из полученного угла и зная радиус (длину стержня маятника), можно найти координаты точки, в которой находится конец маятника в данный момент времени (рис. 1.2).

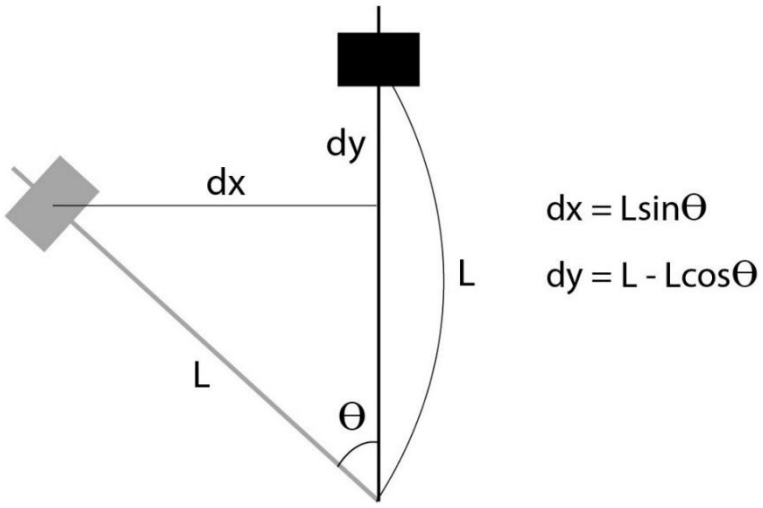

Рис. 1.2: Расчёт координат конца маятника на основе угла отклонения

Достоинство этого метода заключается в достаточно высокой точности вычисления траектории движения маятника. А основным недостатком является достаточно большое количество затратных (тригонометрических) вычислений.

Второй способ моделирования колебательного движения основан на алгоритме поворота тела (маятника) относительно заданной точки (основания маятника). В этом методе конец маятника каждый раз поворачивается вокруг оси Z на определённый угол. После чего, достигнув амплитудного положения, он начинает поворачиваться в обратную сторону (это выражается в изменении угла поворота на противоположный). Значение угла поворота в данном случае прямо-пропорционально зависит от частоты колебания маятника: чем выше темп метронома (частота колебаний маятника), тем больше должно быть значение угла поворота.

Плюсом этого способа являются менее затратные вычисления и, соответственно, более высокая производительность.

Недостатки:

- сложность использования этого метода при повороте модели (в таком случае поворот маятника является более сложным, так как осуществляется не только вокруг оси Z);
- неточность зависимости угла поворота от частоты колебаний, так как не существует конкретных формул.

Таким образом, для моделирования колебаний маятника будет использоваться второй способ, так как он менее затратный с точки зрения вычислений. При этом за амплитуду будет браться максимальный угол отклонения маятника, при котором колебания ещё будут считаться гармоническими (т. е. ~20°).

1.4 Синтез реалистического динамического изображения

1.4.1 Удаление невидимых линий

Изображаемый объект (метроном) удобнее всего представлять в виде двух подобъектов: корпуса (статический объект) и маятника (динамический объект). Оба этих подобъекта являются невыпуклыми телами. Значит, лучше всего будет использовать алгоритмы, для которых неважна выпуклость объектов. Это такие алгоритмы, как: алгоритм Варнока (разбиение окнами), алгоритм, использующий z-буфер, и алгоритм, использующий список приоритетов [5].

В алгоритме Варнока окно делится на подокна до тех пор, пока для каждого окна не будет известно, что нужно изобразить. В простой версии алгоритма окно делится на части, если оно не пусто. Окно является пустым, когда все многоугольники являются внешними по отношению к этому окну. В этом случае предел – 1 пиксель, тогда определяется глубина каждого из рассматриваемых многоугольников в этой точке. Изображается точка

многоугольника, наиболее близкого к пользователю. В более сложных версиях, ставится задача определения, что изображать в очередном окне, размер которого больше 1 пикселя. Самый простой вариант разделения на подокна – это деление каждого окна на 4 равных прямоугольных окна. Этот алгоритм можно оптимизировать сортировкой многоугольников по z, хранением информации (например, если для окна данный многоугольник – охватывающий, то это запоминается и на подокнах уже не проверяется), хранением списков охватывающих, пересекающих и внутренних многоугольников (запоминается, на каком уровне они появились, надо при обходе по дереву из подокон).

В алгоритме, использующем z-буфер, используется идея о буфере кадра. Буфер кадра используется для заполнения атрибутов (интенсивности) каждого пикселя в пространстве изображения. В данном алгоритме используется два буфера: буфер регенерации и сам z-буфер, куда можно помещать информацию о координате z для каждого пикселя. Вначале z-буфер заполняется минимально возможными значениями z, а буфер регенерации – пикселями, описывающими фон. В процессе работы глубина (значение z) каждого нового пикселя, который нужно занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесён в z-буфер. Если это сравнение показывает, что новый пиксель расположен впереди пикселя, находящегося в буфере кадра, то новый пиксель заносится в этот буфер, и производится корректировка z-буфера новым значением z. Если же сравнение даёт противоположный результат, то никаких действий не производится. По сути, алгоритм является поиском по x и y наибольшего значения функции z(x,y).

Достоинства алгоритма:

- сцены могут быть произвольной сложности;
- не требуется сортировка;
- трудоёмкость линейно зависит от числа рассматриваемых поверхностей.

Основной недостаток алгоритма заключается в использовании большого объёма памяти.

Алгоритм, использующий список приоритетов, пытается получить преимущество предварительной сортировкой по глубине. На выходе получается окончательный список элементов сцены, упорядоченных по приоритету глубины, основанному на расстоянии от точки наблюдения. Сначала формируется предварительный список приоритетов по глубине из всех многоугольников. В качестве ключа сортировки используется z_{min}. Затем для каждого многоугольника (P) из списка проверяются отношения с другими более ближними многоугольниками (Q): если ближайшая вершина P находится дальше от наблюдателя, чем самая удалённая вершина Q, то никакая часть P не может экранировать Q. В ином случае P может потенциально экранировать многоугольники типа Q. Также могут быть случаи циклического экранирования (рис. 1.5).

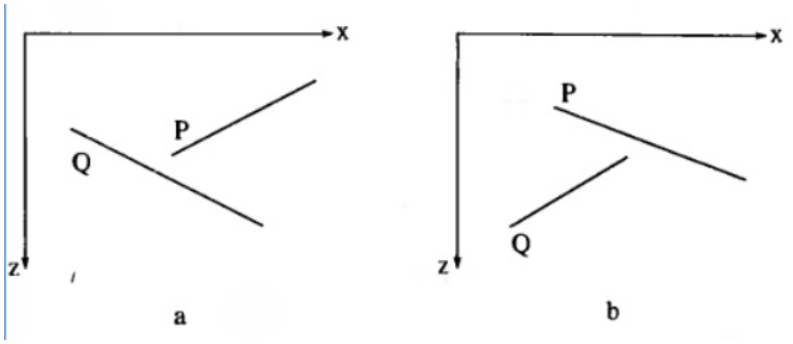

Рис. 1.4: Случаи, когда P мог бы перекрывать Q, но не перекрывает

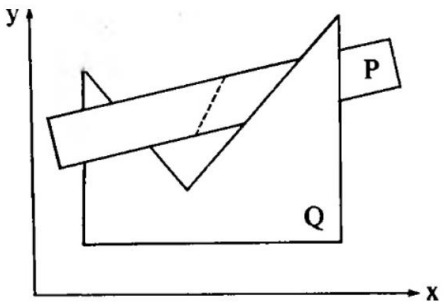

Рис. 1.5: Пример циклического экранирования

Таким образом, для удаления невидимых линий и поверхностей будет использоваться алгоритм, использующий z-буффер, так как он является наиболее «стабильным» и справляется со сценами любой сложности, в отличие от, например, алгоритма со списком приоритетов.

1.4.2 Модель освещения

Для достижения наиболее высокой производительности будет использоваться локальная модель освещения с двумя составляющими интенсивности: фоновая освещённость (I_{amb}) и диффузное отражение (I_{diff}).

Фоновое освещение – это постоянная в каждой точке величина надбавки к освещению. Вычисляется следующим образом [5]:

$$I_{amb} = k_a \cdot I_a, \tag{1.7}$$

где k_a – коэффициент в пределах от 0 до 1, характеризующий отражающие свойства поверхности, I_a – исходная интенсивность освещения, которое падает на поверхность.

$$I_{diff} = k_{diff} \cdot I_d \cdot \cos\theta, \tag{1.8}$$

где k_a – коэффициент в пределах от 0 до 1, характеризующий рассеивающие свойства поверхности, I_d – интенсивность падающего на поверхность света, $\cos(\theta)$ – угол между направлением на источник света и нормалью поверхности.

С учётом и фоновой освещённости, и диффузного отражения получается более реалистичное изображение.

Таким образом, конечная формула интенсивности имеет вид:

$$I = k_{diff} \cdot I_d \cdot \cos\theta + k_a \cdot I_a, \tag{1.9}$$

1.4.3 Методы закраски поверхностей

Есть три возможных метода закрашивания [5]: простая закраска, закраска по Гуро и закраска по Фонгу. У метронома все его поверхности являются плоскостями, поэтому закраску по Фонгу, предполагающую улучшение

аппроксимации кривизны поверхности, использовать нет смысла. Простую закраску (одна плоскость закрашивается одним уровнем интенсивности) также использовать не получится, так как в нашем случае источник света и наблюдатель будут находиться не в бесконечности. Значит, будет использоваться метод Гуро.

В закраске методом Гуро сначала вычисляются нормали к вершинам граней. Если известны уравнения плоскостей, содержащих грани, то для нахождения нормалей используется формула:

$$n = (a_0 + a_1 + a_2)i + (b_0 + b_1 + b_2)j + (c_0 + c_1 + c_2)k, \qquad (1.10)$$

где a_0, b_0, c_0, d_0, a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2 – коэффициенты уравнений плоскостей трёх многоугольников, окружающих вершину, в которой ищется нормаль.

Если же уравнения плоскостей не заданы, то нормаль к вершине определяется усреднением векторных произведений всех рёбер, пересекающихся в этой вершине:

$$n_{V_1} = V_1V_2 \times V_1V_4 + V_1V_5 \times V_1V_2 + V_1V_4 \times V_1V_5, \qquad (1.11)$$

где V_1V_2, V_1V_4, V_1V_5 – рёбра, пересекающиеся в вершине V_1, в которой ищется значение нормали (рис. 1.6).

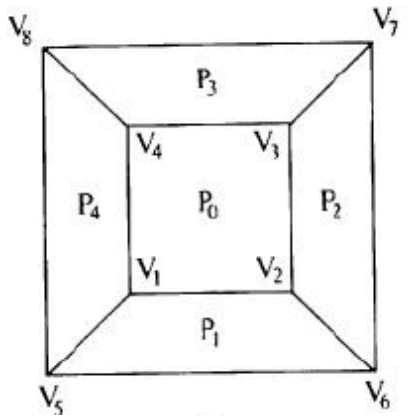

Рис. 1.6: Аппроксимация нормали к полигональной поверхности

После этого, зная нормаль, вычисляется интенсивность каждой вершины и выполняется первая линейная интерполяция вдоль рёбер. Интенсивность вершин вычисляется по формуле:

$$I = I_{\text{р}} k_{\text{р}} + \frac{I_{\text{и}}}{d+K}\left[k_{\text{д}} \frac{\bar{\boldsymbol{n}} \cdot \bar{L}}{|\bar{\boldsymbol{n}}||\bar{L}|} + k_{\text{з}} \left(\frac{R \cdot S}{|R||S|}\right)^{n}\right], \quad (1.12)$$

где: $I_{\text{р}}$, $I_{\text{и}}$ – интенсивности рассеянного света и источника;

$k_{\text{р}}$, $k_{\text{д}}$, $k_{\text{з}}$ – коэффициенты рассеянного света и диффузного и зеркального отражений;

d – расстояние от центра проекции до объекта;

К – произвольная постоянная, выбираемая на основе эксперимента;

$\bar{\boldsymbol{n}}$, $\bar{L}$ – векторы нормали и падения света; R, S – векторы отражения и наблюдения;

n – степень, аппроксимирующая пространственное распределение зеркально отражённого света.

Вторая линейная интерполяция выполняется при вычислении интенсивностей пикселей, расположенных на сканирующей строке. Значения интенсивностей вдоль сканирующей строки можно вычислять инкрементально:

$$I_{P_2} = I_{P_1} + \left(I_Q - I_R\right)(t_2 - t_1) = I_{P_1} + \Delta I \Delta t, \quad (1.13)$$

где t_1, t_2 – расположение двух пикселей на сканирующей строке, Q, R – точки пересечения рёбер многоугольника со сканирующей строкой.

Недостаток этого метода закраски в том, что можно потерять рёбра и получить плоское изображение. Это может произойти, когда закрашивается смежная грань одним уровнем интенсивности. Решить эту проблему можно, немного изменив некоторые значения нормалей.

1.4.4 Учёт теней

В данной задаче источник освещения будет точечный, то есть создающий только полную тень. Также источник будет располагаться на конечном

расстоянии, но вне поля зрения наблюдателя, что означает, что для построения теней будет использоваться перспективная проекция.

Алгоритм построения собственных теней [5] аналогичен алгоритму удаления нелицевых граней: грани, затенённые собственной тенью, являются нелицевыми, если точку наблюдения совместить с источником света.

Чтобы найти проекционные тени, нужно построить проекции всех нелицевых граней на сцену. Центр проекции – источник света. Точки пересечения проецируемой грани со всеми другими плоскостями образуют многоугольники, которые помечаются как теневые и заносятся в структуру данных (многоугольник на горизонтальной плоскости на рис. 1.7). Чтобы не вносить в неё слишком много многоугольников, можно проецировать только контур каждого объекта, а не отдельные грани.

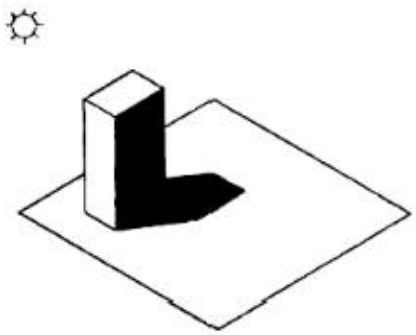

Рис. 1.7: Тени

Стоит отметить, что для разных точек наблюдения нет смысла каждый раз пересчитывать тени, так как они изменятся только при изменении источника света.

1.4.5 Трансформирование объектов в трёхмерном пространстве

Матрица преобразований в трёхмерном пространстве будет иметь размерность 4x4. Координаты точек (x, y, z) заменятся четвёркой (wx, wy, wz, w), w != 0.

Для переноса используется матрица:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{pmatrix} \quad (1.14)$$

Для масштабирования:

$$\begin{pmatrix} kx & 0 & 0 & 0 \\ 0 & ky & 0 & 0 \\ 0 & 0 & kz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.15)$$

Для поворота вокруг ox:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & sin\theta & 0 \\ 0 & -sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.16)$$

Для поворота вокруг oy:

$$\begin{pmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.17)$$

Для поворота вокруг oz:

$$\begin{pmatrix} cos\theta & sin\theta & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.18)$$

При использовании матриц масштабирования и поворота сначала нужно перенести тело и центр масштабирования (поворота) так, чтобы этот центр оказался в начале координат.

Выводы

В данном разделе был проведён анализ возможных алгоритмов моделирования колебаний и синтеза реалистического изображения. В результате чего:

- для моделирования колебаний выбран алгоритм, основанный на повороте тела относительно заданной точки;

- для удаления невидимых линий и поверхностей выбран алгоритм, использующий z-буффер;
- для закраски выбран метод Гуро;
- выбрана простая (локальная) модель освещения (т. е. без использования вторичных источников освещения).

Входными данными для разрабатываемого ПО будут являться:

- файлы формата obj с описанием корпуса и маятника метронома;
- координаты и интенсивность источника света;
- темп метронома;
- коэффициенты масштабирования, переноса и поворота метронома.

Выходным данным является трёхмерное изображение метронома.

Ограничения, в рамках которых будет работать программа:

- минимально возможное значение темпа метронома – 20 bpm. Максимальное – 400 bpm;
- метроном можно запустить, только если он находится в исходном положении (т. е. не повёрнут и не перемещён).

Функциональные требования к разрабатываемому ПО:

- на экран должна выводиться трёхмерная реалистическая модель метронома;
- должна быть возможность запуска и остановки метронома;
- должна быть возможность изменения темпа метронома;
- должна быть возможность масштабирования, переноса и поворота модели;
- должна быть возможность переноса и поворота камеры;
- должна быть возможность добавления нового источника света;
- должна быть возможность изменения положения и интенсивности источника света.

2 Конструкторская часть

В этом разделе на основе теоретических данных, полученных в аналитическом разделе, будут построены схемы разрабатываемых алгоритмов. А также будут приведены: структуры данных, способы и этапы тестирования, память, используемая программой, и диаграмма классов.

2.1 Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма с использованием z-буффера.

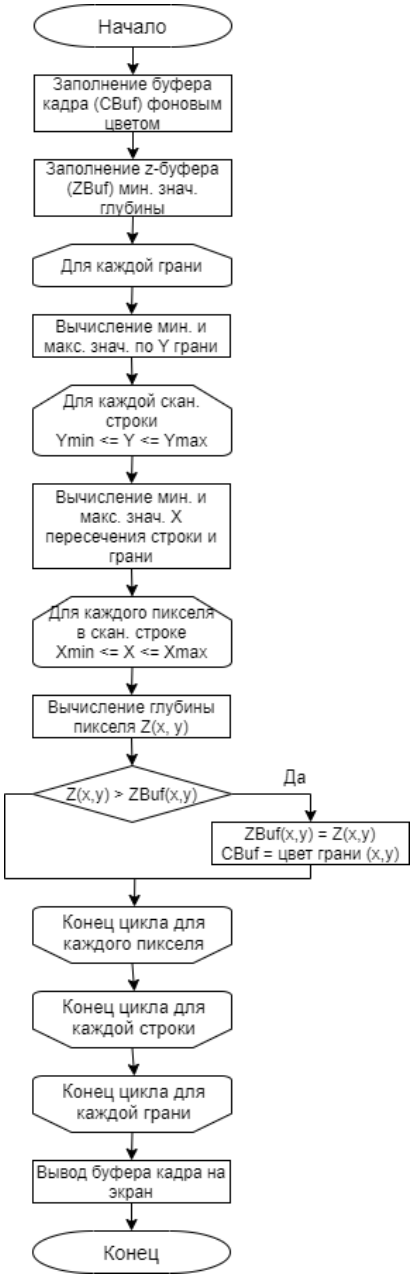

Рис. 2.1: Схема алгоритма z-буффера

На рисунке 2.2 представлена схема алгоритма закраски по Гуро.

Рис. 2.2: Схема алгоритма закраски по Гуро

На рисунке 2.3 представлена схема алгоритма поворота конца маятника.

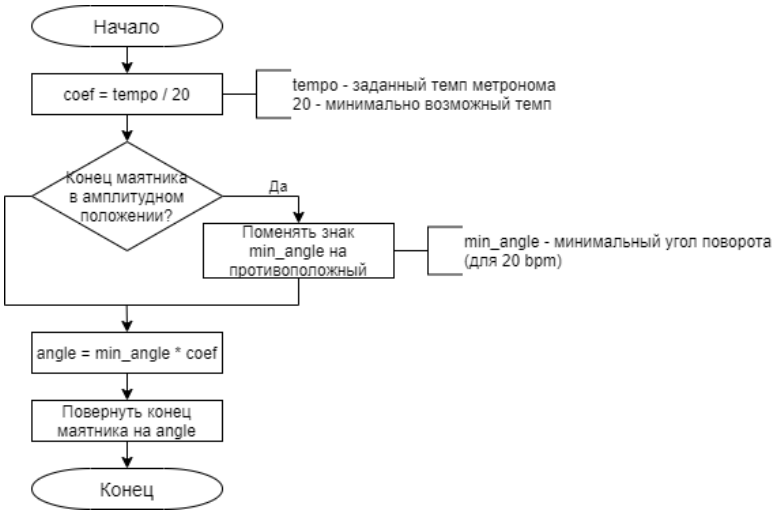

Рис. 2.3: Схема алгоритма поворота конца маятника

2.2 Структуры данных

В основном в программе для представления различных данных будут использоваться векторы, так как они наиболее удобны для добавления и удаления и имеют произвольный доступ.

Вершины модели и нормали к её граням будут храниться в виде векторов, элементами которых являются трёхмерные координаты. А сами грани будут храниться в виде векторов, элементами которых являются подсписки, состоящие из индексов, которые ссылаются на соответствующие вершины.

Z-буфер и буфер кадра также будут представляться в виде двумерного вектора. А источники света будут храниться в одномерном векторе.

2.3 Способы и этапы тестирования

Для проверки работоспособности ПО будет применяться функциональное тестирование.

Тестирование ПО будет разделено на следующие этапы:

- тестирование функции запуска метронома при различных значениях темпа;
- тестирование функций поворота, перемещения и масштабирования метронома;
- тестирование функций поворота и перемещения камеры;
- тестирование функции запуска метронома при повороте и/или перемещении камеры;
- тестирование функции запуска метронома при его повороте, перемещении и/или масштабировании;
- тестирование функций поворота, перемещения и масштабирования метронома в работающем состоянии;
- тестирование функций добавления и изменения параметров источника света.

2.4 Используемая память

Так как в программе в основном используются динамические структуры данных (векторы), то нужно будет контролировать утечки памяти.

Количество памяти, необходимой программе для хранения данных, прямо пропорционально длинам векторов.

2.5 Диаграмма классов

На рисунке 2.4 приведена диаграмма классов.

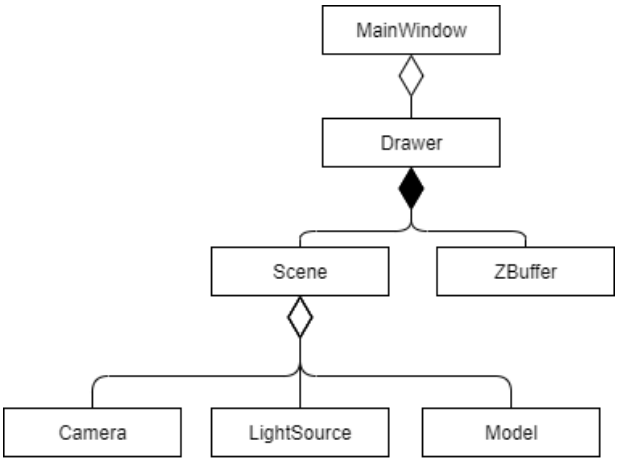

Рис. 2.4: Диаграмма классов

Разрабатываемая программа будет состоять из следующих классов:

- Model – класс трёхмерных объектов (корпуса и матяника) с возможностью перемещения, масштабирования и поворота вокруг собственного центра;
- Camera – класс камеры с возможностью перемещения и поворота;
- LightSource – класс источника освещения с возможностью перемещения по сцене и изменения интенсивности;
- Scene – класс, который содержит в себе все объекты сцены;
- ZBuffer – класс, который предоставляет функционал для работы с буфером глубины;
- Drawer – класс-менеджер, который связывает сцену и интерфейс системы;

- MainWindow – основной класс, который предоставляет интерфейс пользователю.

Выводы

На основе теоретических данных, полученных в аналитическом разделе, были построены схемы разрабатываемых алгоритмов. А также были приведены: структуры данных, способы и этапы тестирования, память, используемая программой, и диаграмма классов.

3 Технологическая часть

В этом разделе будет разработан и протестирован исходный код программного обеспечения.

3.1 Средства реализации

Для реализации проекта в качестве языка программирования был выбран язык C++ [5] – компилируемый, статически типизируемый язык программирования общего назначения. Поддерживает такие парадигмы программирования, как процедурное программирование, объектно-ориентированное программирование и обобщённое программирование, также обеспечивает модульность, раздельную компиляцию, обработку исключений, абстракцию данных, объявление типов (классов) объектов, виртуальные функции. Стандартная библиотека включает в себя некоторые уже готовые контейнеры и алгоритмы.

В качестве среды разработки была выбрана среда «Qt Creator» [6] – кроссплатформенная свободная IDE для разработки на C, C++, JavaScript и QML. Включает в себя графический интерфейс отладчика и визуальные средства разработки интерфейса как с использованием QtWidgets, так и QML.

3.2 Детали реализации

В листинге 3.1 представлена реализация преобразования мировых координат вершин модели в репер камеры. Этот метод также вызывает функции расчёта интенсивности и растеризации полигона.

Листинг 3.1: Метод преобразования мировых координат в репер камеры

```
void DrawerobjectProcessing(Model& model, Vector3f& camPos, Vector3f&
camDir, Vector3f& camUp)
{
    size_t i, j;
    bool skip;
    float camZInc = fabs(camPos.z) + 1;
    float camZDec = fabs(camPos.z) - 1;

    Vector3f center = model.getCenter();
    size_t faces    = model.getFacesCount();
    QColor color    = model.getColor();
```

```
    Matrix viewPort   = Cameraviewport(w8, h8, w34, h34);
    Matrix projection = Matrixidentity(4);
    Matrix modelView  = CameralookAt(camPos, camDir, camUp);

    projection[3][2]  = - 1.f  (camPos - camDir).norm();

    Matrix mvp = viewPort  projection  modelView;

    for (i = 0; i  faces; i++)
    {
        skip = false;
        stdvectorint face = model.face(i);

        Vector3i screenCoords[3];
        float    intensity[3] = { BG_LIGHT, BG_LIGHT, BG_LIGHT };

        for (j = 0; j  3; j++)
        {
            Vector3f v = center + model.vert(face[j]);

            if (v.z  camZDec && v.z  camZInc)
            {
                skip = true;
                break;
            }

            screenCoords[j] = Vector3f(mvp  Matrix(v));
            intensity[j] = lightProcessing(v, model.norm(i, j));
        }

        if (skip || !checkIsVisible(screenCoords[0]) ||
                    !checkIsVisible(screenCoords[1]) ||
                    !checkIsVisible(screenCoords[2])) continue;

        triangleProcessing(screenCoords[0], screenCoords[1],
screenCoords[2],
                color, intensity[0], intensity[1], intensity[2]);
    }
}
```

В листинге 3.2 представлена реализация метода, который рассчитывает интенсивность вершины от источников света на сцене.

Листинг 3.2: Расчёт интенсивности вершины от источников света

```
float Drawer::lightProcessing(const Vector3f& vert, const Vector3f& norm)
{
    float wholeIntensity = 0;
    float intensity;

    size_t lights = scene.getLightSourceCount();

    for (size_t i = 0; i < lights; i++)
    {
        intensity = 0;
        LightSourcePoint lsp = scene.getLightSource(i);

        Vector3f lightDir = vert - lsp.getPosition();

        intensity += lightDir * norm / pow(lightDir.norm(), 2.0);
        intensity *= lsp.getIntensity() * LIGHT_REFLECT;
```

```
        intensity = fmax(0.0, intensity);
        intensity = fmin(1.0, intensity);

        intensity = BG_LIGHT + intensity * (1 - BG_LIGHT);

        wholeIntensity += intensity;
    }

    if (wholeIntensity == 0)
        wholeIntensity = BG_LIGHT;
    else
        wholeIntensity /= lights;

    return wholeIntensity;
}
```

В листинге 3.3 представлена реализация метода, который вычисляет глубину и интенсивность каждой точки полигона.

Листинг 3.3: Вычисление глубины и интенсивности точек полигона

```
void Drawer::triangleProcessing(Vector3i& t0, Vector3i& t1, Vector3i& t2,
                                const QColor& color, float& i0, float& i1,
float& i2)
{
    if (t0.y == t1.y && t0.y == t2.y)
        return;

    if (t0.y > t1.y)
    {
        std::swap(t0, t1);
        std::swap(i0, i1);
    }
    if (t0.y > t2.y)
    {
        std::swap(t0, t2);
        std::swap(i0, i2);
    }
    if (t1.y > t2.y)
    {
        std::swap(t1, t2);
        std::swap(i1, i2);
    }

    int total_height = t2.y - t0.y;

    for (int i = 0; i < total_height; i++)
    {
        bool second_half = i > t1.y - t0.y || t1.y == t0.y;
        int segment_height = second_half ? t2.y - t1.y : t1.y - t0.y;

        float alpha = (float)i / total_height;
        float betta = (float)(i - (second_half ? t1.y - t0.y : 0)) /
segment_height;

        Vector3i A =               t0 + Vector3f(t2 - t0) * alpha;
        Vector3i B = second_half ? t1 + Vector3f(t2 - t1) * betta : t0 +
Vector3f(t1 - t0) * betta;

        float iA =              i0 + (i2 - i0) * alpha;
```

```
        float iB = second_half ? i1 + (i2 - i1) * betta : i0 + (i1 - i0) *
betta;

        if (A.x > B.x)
        {
            std::swap(A, B);
            std::swap(iA, iB);
        }

        A.x = std::max(A.x, 0);
        B.x = std::min(B.x, w);

        for (int j = A.x; j <= B.x; j++)
        {
            float phi = B.x == A.x ? 1. : (float)(j - A.x) / (float)(B.x -
A.x);

            Vector3i P = Vector3f(A) + Vector3f(B - A) * phi;
            float iP = iA + (iB - iA) * phi;

            if (P.x >= w || P.y >= h || P.x < 0 || P.y < 0) continue;

            if (zBuffer.getDepth(P.x, P.y) < P.z)
            {
                zBuffer.setDepth(P.x, P.y, P.z);
                colorCache[P.x][P.y] = QColor(iColor(color.rgba(), iP));
            }
        }
    }
}
```

3.3 Интерфейс программы

На рисунке 3.1 представлен интерфейс основного окна программы.

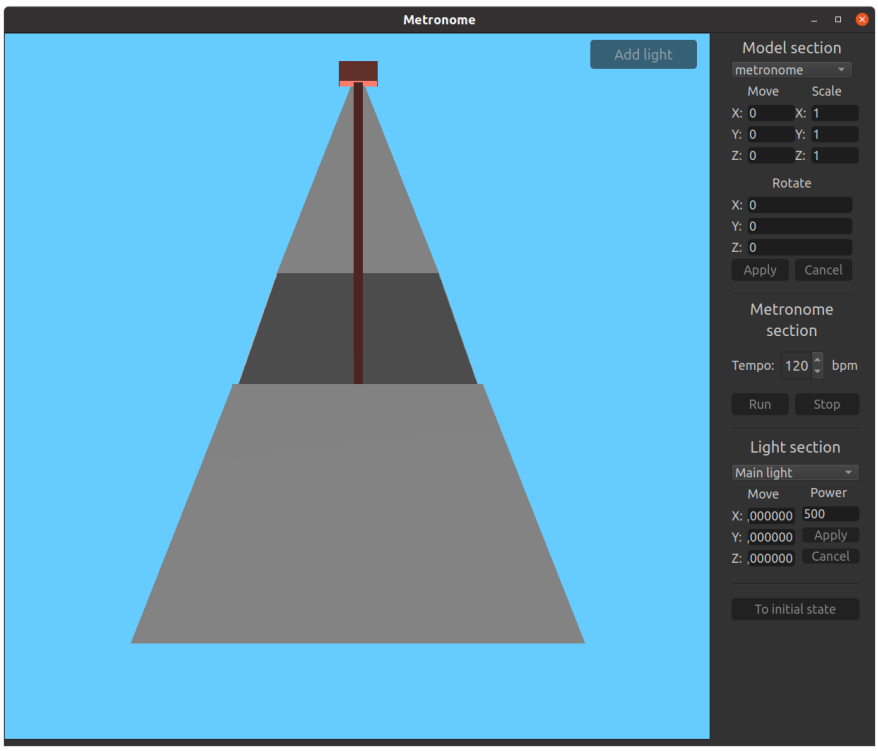

Рис. 3.1: Интерфейс основного окна программы

Для управления моделью используется панель справа. Для взаимодействия с камерой используется набор клавиш wasd (для передвижения по сцене вперёд, влево, назад, вправо соответственно) и ijkl (для изменения направления взгляда камеры вверх, влево, вниз, вправо соответственно).

Для добавления источников света используется кнопка “Add light” в правом верхнем углу. При нажатии на эту кнопку появляется новое окно (рис. 3.2), в котором можно задать имя добавляемого источника света, его координаты и интенсивность.

Рис. 3.2: Интерфейс окна добавления нового источника света

В правом нижнем углу основного окна программы можно изменять параметры существующих источников света, выбрав в выпадающем списке соответствующий источник света.

3.4 Тестирование ПО

Ниже приведены результаты тестирования работоспособности ПО. Как видно по результатам, все тесты были успешно пройдены, так как выдают предполагаемые результаты.

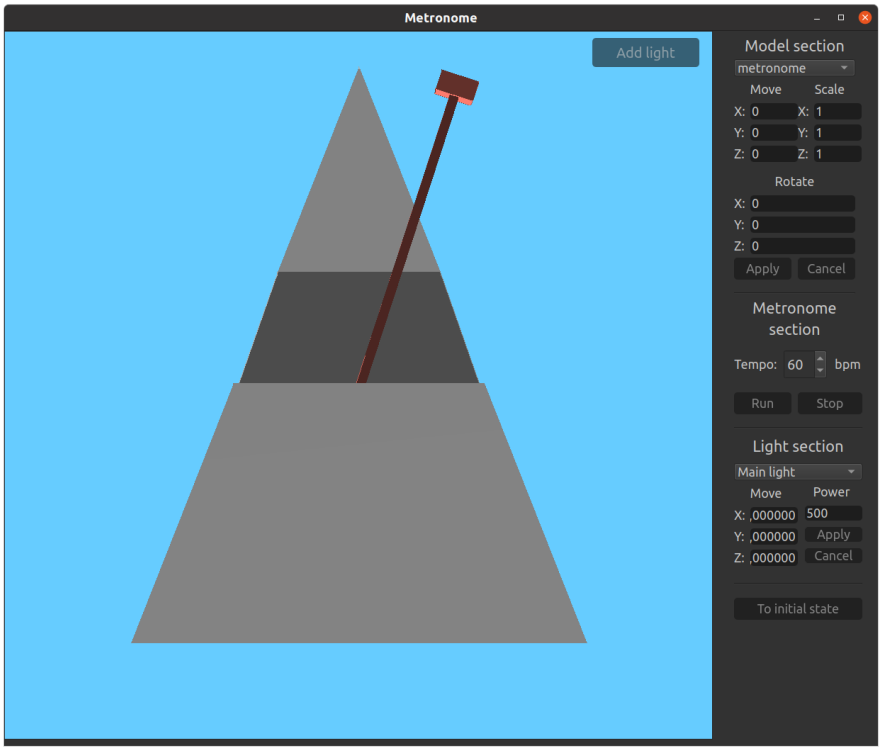

Рис. 3.3: Тестирование запуска при разных значениях темпа

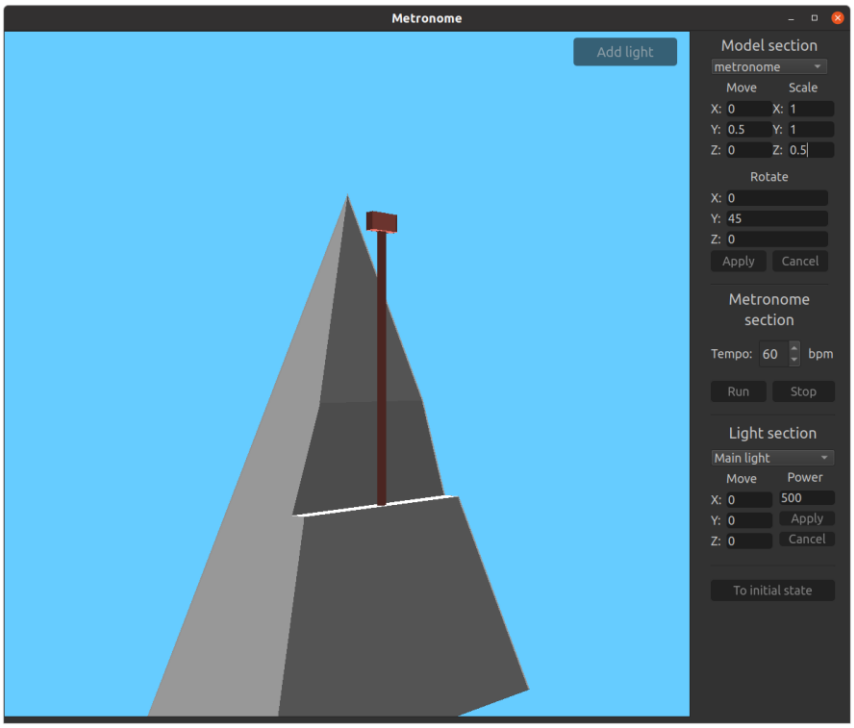

Рис. 3.5: Тестирование преобразования модели

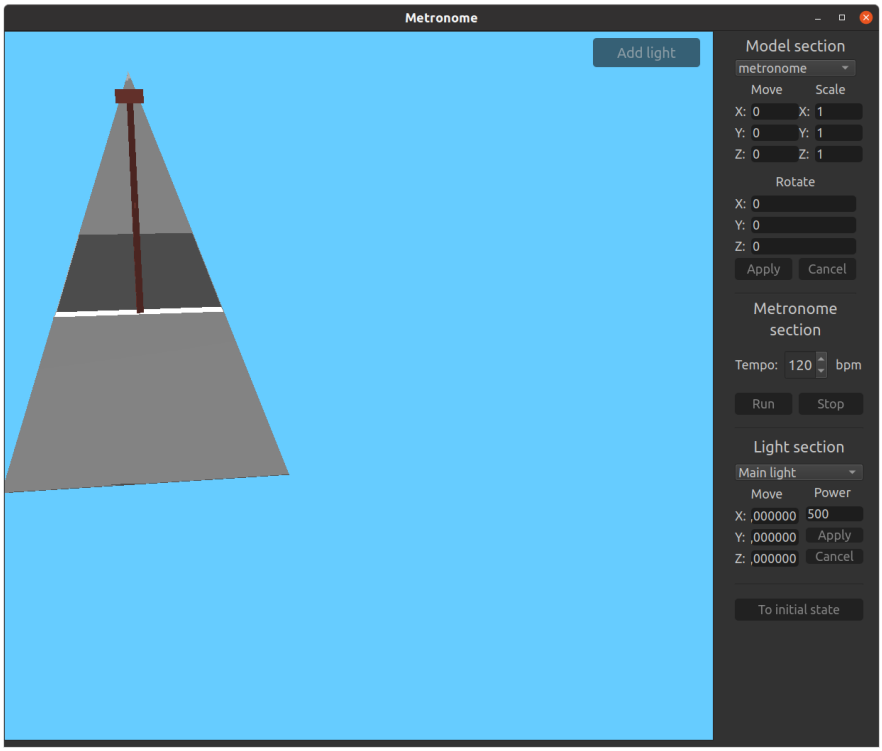

Рис. 3.7: Тестирование преобразования камеры

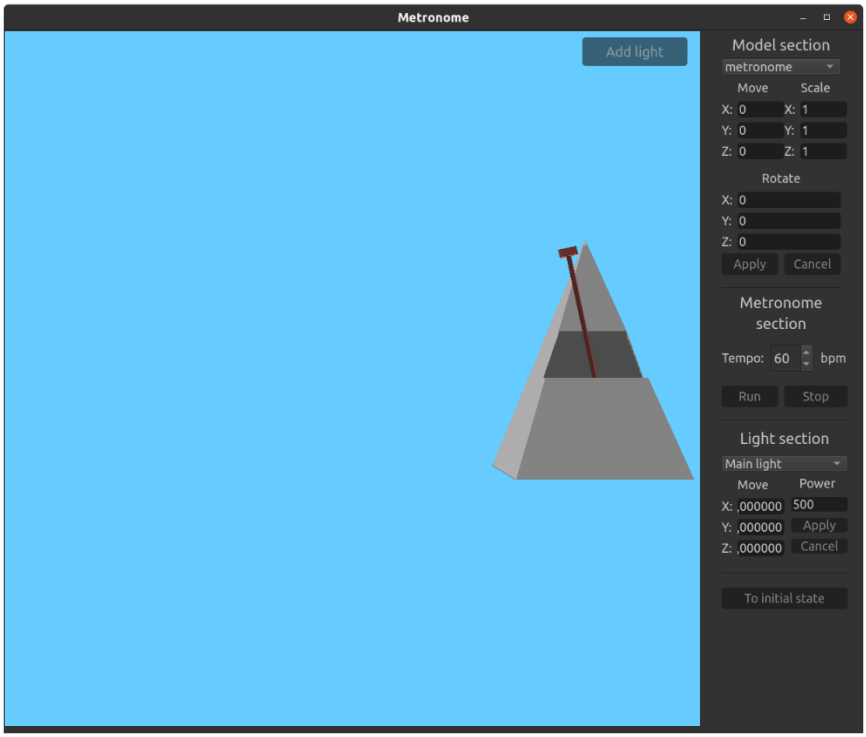

Рис. 3.8: Тестирование запуска при преобразовании камеры

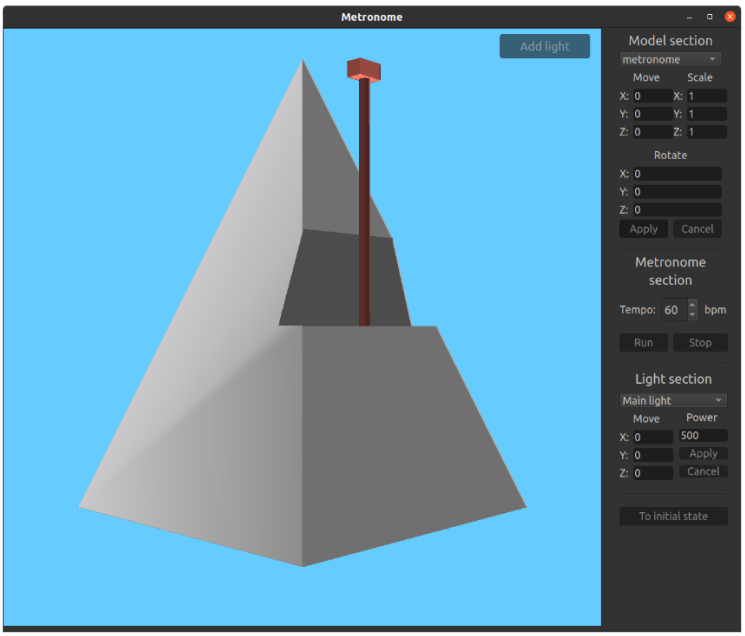

Рис. 3.9: Тестирование добавления источника света

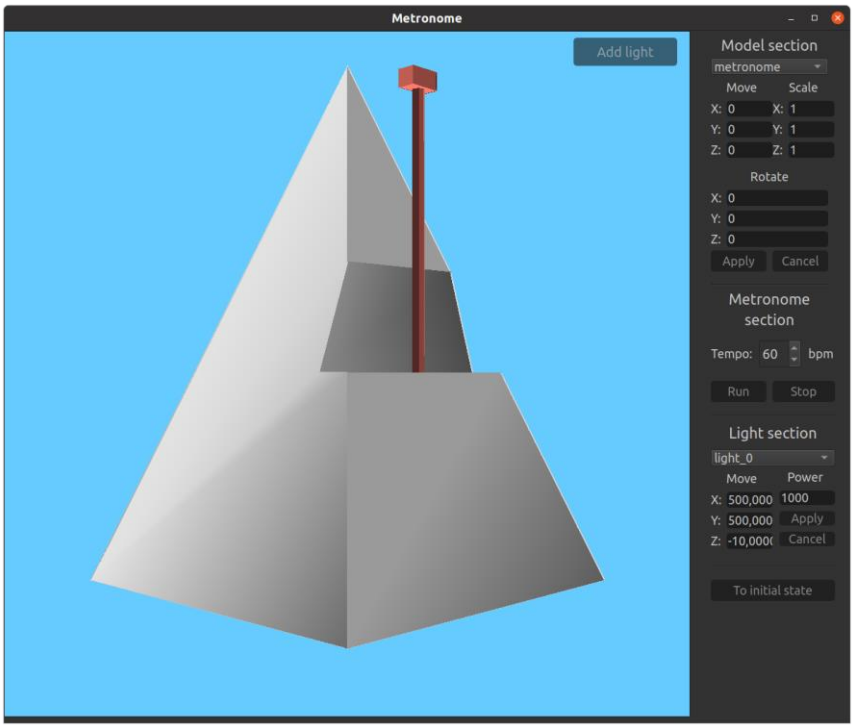

Рис. 3.10: Тестирование изменения параметров источника света

Выводы

В данном разделе был разработан и протестирован исходный код программного обеспечения. Как видно выше, все тесты были успешно пройдены, так как дали ожидаемые результаты.

4 Исследовательская часть

В этом разделе будет проведён анализ времени отрисовки модели в зависимости от числа её вершин.

4.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором был проведён эксперимент:

- операционная система: Ubuntu 20.04.3 LTS [7];
- оперативная память: 8 GB;
- процессор: 11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz [8].

4.2 Постановка эксперимента

Измерения проводятся при одном источнике освещения, который для всех тестов находится в одной позиции. На сцену поочерёдно загружаются модели с количеством вершин: 160, 473, 763, 1258, 2050, 2264. Каждый замер проводится 10 раз, при этом в качестве результата выбирается среднее арифметическое полученных замеров времени.

4.3 Результаты эксперимента

По результатам измерения времени можно составить таблицу 4.1 и диаграмму 4.1.

Таблица 4.1: Результаты замеров времени отрисовки сцены

Количество вершин	160	473	763	1258	2050	2264
Среднее время отрисовки в микросекундах	40156	83974	96865	108196	178467	354619

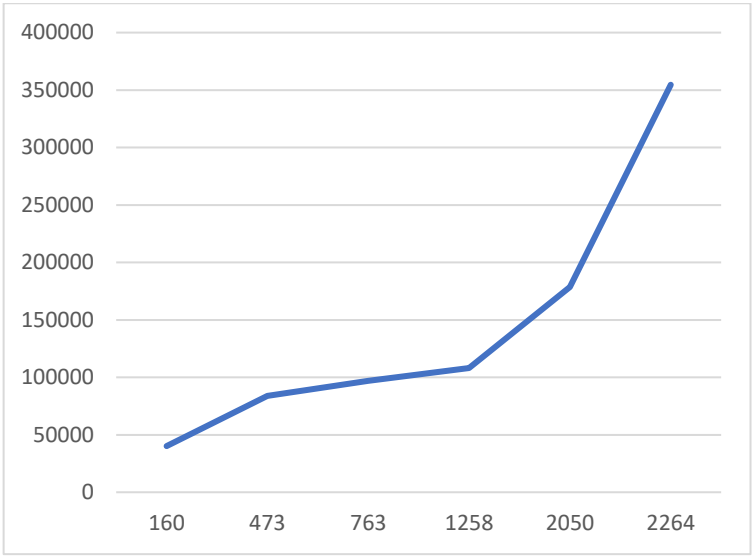

Диаграмма 4.1: Зависимость времени отрисовки от количества вершин

Выводы

В данном разделе был проведён анализ времени отрисовки модели в зависимости от числа её вершин.

Было выяснено, что увеличение количества вершин значительно влияет на скорость отрисовки изображения. Чем больше количество вершин, тем больше времени тратится на отрисовку сцены.

Заключение

Цель курсовой работы достигнута. Спроектировано и реализовано программное обеспечение для визуализации работы метронома.

В ходе работы были проанализированы существующие алгоритмы удаления невидимых линий и поверхностей, модели освещения, закраски и указаны их преимущества и недостатки. После чего были разработаны алгоритмы, необходимые для решения поставленной задачи.

Разработанный продукт является достаточно хорошим для его дальнейшего использования музыкантами во время занятий и репетиций.

Список литературы

[1] Формулы математического маятника. URL: https://www.webmath.ru/poleznoe/fizika/fizika_149_formuly_matematicheskogo_majatnika.php. Дата обращения: 10.07.2021.

[2] Свободные колебания. Математический маятник. URL: https://physics.ru/courses/op25part1/content/chapter2/section/paragraph3/theory.html#.YOLtUegzZPY. Дата обращения: 10.07.2021.

[3] Прохоров А. М. Маятник // Физический энциклопедический словарь. — М.: Советская энциклопедия. — 1983.

[4] Косников Ю. Н. Поверхностные модели в системах трёхмерной компьютерной графики. Учебное пособие. – Пенза: Пензенский государственный университет, 2007. – 60 с.

[5] Роджерс Д. Алгоритмические основы машинной графики. – М., «Мир», 1989.

[6] Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином-Пресс, 2007. — 1104 с.

[7] Category:Tools::QtCreator. URL: https://wiki.qt.io/Category:Tools::QtCreator. Дата обращения: 12.01.2022.

[8] Ubuntu 20.04.3 LTS (Focal Fossa). URL: https://releases.ubuntu.com/20.04/. Дата обращения: 21.01.2022.

[9] Процессор Intel® Core™ i3-1115G4 (6 МБ кэш-памяти, до 4,10 ГГц). URL: https://www.intel.ru/content/www/ru/ru/products/sku/208652/intel-core-i31115g4-processor-6m-cache-up-to-4-10-ghz/specifications.html. Дата обращения: 21.01.2022.