Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Новосибирский государственный технический университет»

Кафедра прикладной математики

**Отчет ПО ПРАКТИКЕ**

Производственная практика: преддипломная практика

(наименование практики в соответствии с учебным планом)

Направление подготовки: 01.03.02 Прикладная математика и информатика

Профиль: Компьютерное моделирование и информационные технологии.

|  |  |
| --- | --- |
| Выполнил:  Студент Грибова А.П.  *(Ф.И.О.)*  Группа ПМ-13  Факультет ПМИ .  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *подпись*  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. | Проверил:  Руководитель от НГТУ Задорожный А.Г.  *(Ф.И.О.)*  Балл: \_\_\_\_\_\_\_\_\_\_, ECTS\_\_\_\_\_\_\_\_\_\_\_\_\_,  Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «отлично», «хорошо», «удовлетворительно», «неуд.», «зачтено», «не зачтено»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *подпись*  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г. |
|  |  |

Новосибирск 2025

**Содержание**

[Введение 3](#_Toc198729005)

[1. Теоретическая часть 4](#_Toc198729006)

[1.1. Поставленные задачи 4](#_Toc198729007)

[1.2. Библиотека OpenGL 5](#_Toc198729008)

[1.2.1. Матрицы преобразований 5](#_Toc198729009)

[1.2.2. Модель сферы 7](#_Toc198729010)

[1.3. Реализация движения маятника 8](#_Toc198729011)

[2. Описание разработанной программы 11](#_Toc198729012)

[2.1. Освещение 11](#_Toc198729013)

[2.2. Настройка сцены 11](#_Toc198729014)

[2.3. Цикл отрисовки 12](#_Toc198729015)

[2.3. Освещение 13](#_Toc198729016)

[2.4. построение сферы 14](#_Toc198729017)

[3.2. Интерфейс программы 15](#_Toc198729018)

[3. Примеры работы программы 18](#_Toc198729019)

[3.1. Тест №1 18](#_Toc198729020)

[3.2. Тест №2 21](#_Toc198729021)

[3.3. Тест №3 23](#_Toc198729022)

[3.4. Тест №4 25](#_Toc198729023)

[3.5. Тест №5 27](#_Toc198729024)

[Заключение 31](#_Toc198729025)

[Список литературы 32](#_Toc198729026)

[Приложение. Код программы 33](#_Toc198729027)

Введение

Целью данного проекта является разработка программного обеспечения, которое позволит визуализировать физическую симуляцию движения маятника в реалистичном трехмерном пространстве. Проект направлен на создание программы, объединяющей точное математическое моделирование физических процессов с красочным и интуитивным графическим интерфейсом.

Основное внимание в проекте уделяется реализации физической модели, которая максимально точно воспроизводит динамику движения маятника. При этом особое значение придается интеграции этой симуляции с трехмерной графикой, что позволяет создать визуально насыщенную и привлекательную сцену.

Важным аспектом работы является использование современных технологий трехмерной графики, включая настройку источников света, реалистичный расчет освещения, и точную отрисовку теней, создающих глубину и объемность сцены. Все эти элементы обеспечивают более высокий уровень визуальной достоверности и создают эффект полного погружения.

Для удобства пользователей в программу планируется добавить интерфейс, позволяющий изменять ключевые параметры сцены и объектов. Такой функционал обеспечит гибкость и наглядность работы, позволяя пользователю, например, регулировать длину маятника, силу гравитации, интенсивность освещения или положение источников света.

Этот проект представляет собой синтез науки и искусства, где точные вычисления и технологии трехмерной графики объединяются для создания не только функционального, но и эстетически впечатляющего продукта.

. Теоретическая часть

Маятник является простейшей механической системой, представляющей собой тело на невесомой нерастяжимой нити, закреплённой на неподвижной точке. Визуализация движения маятника позволяет продемонстрировать базовые законы механики и физики, такие как силы тяжести, момент силы и затухание.

Визуализация движения маятника будет выполняться с использованием библиотеки OpenGL.

1.1. Поставленные задачи

Были поставлены следующие задачи:

• Реализовать движение маятника с помощью OpenGL

• Освещение на маятник падает из одной точки (при движении шарика, блик на нем меняет местоположение)

• Наличие различных источников света

• Динамическая тень, которую отбрасывает шарик маятника

Маятник моделируется как шарик, подвешенный на стержне или нитке. Его движение можно описать как колебания вокруг точки подвеса. В OpenGL это можно реализовать с помощью изменения угла наклона маятника с течением времени и применения вращений для позиционирования элементов сцены.

Прожектор можно представить, как источник света, который освещает сцену в определенном направлении, создавая эффект узкого светового луча. Такой прожектор можно разместить в любой части сцены, и его свет будет воздействовать только на объекты, попадающие в зону его действия. Этот источник света также имеет фиксированное направление и угол покрытия.

Тень, которую отбрасывает шарик маятника, меняется в зависимости от его положения и положения источников света.

1.2. Библиотека OpenGL

OpenGL (Open Graphics Library) – это открытая графическая библиотека для работы с двумерной и трехмерной графикой.

Основным принципом работы OpenGL является получение наборов векторных графических примитивов в виде точек, линий и треугольников с последующей математической обработкой полученных данных и построением растровой картинки на экране. Абсолютное большинство команд OpenGL попадает в одну из двух групп: либо они добавляют графические примитивы на вход в конвейер, либо конфигурируют конвейер на различное исполнение трансформаций.

OpenGL является низкоуровневым процедурным API, что вынуждает программиста диктовать точную последовательность шагов, чтобы построить результирующую растровую графику. С одной стороны, императивный подход требует от программиста глубокого знания законов трёхмерной графики и математических моделей, с другой стороны – даёт свободу внедрения различных инноваций.

1.2.1. Матрицы преобразований

Существуют следующие трехмерные аффинные преобразования:

1. Масштабирование.

2. Поворот.

3. Перенос.

Для получения матриц этих преобразований перейдем в однородную систему координат размерности 4:

. (1)

В матричном виде преобразование любой точки, являющейся вектором, рассчитывается через произведение таких матриц:

(2)

Масштабирование, вызываемое функцией *glm::Scale(x,y,z)*, описывается следующей матрицей:

(3)

Поворот на угол *a*, вызываемый функцией *glm:: Rotate(a,x,y,z)*, матрицей, имеющей сложную структуру, поэтому поворот обычно рассматривается через элементарные повороты:

1. Поворот относительно *Ox* (команда *glm::Rotate(a,1,0,0)*):

(4)

2. Поворот относительно *Oy (*команда *glm::Rotate(a,0,1,0))*:

(5)

3. Поворот относительно *Oz (*команда *glm::Rotate(a,0,0,1))*:

(6)

Чтобы определить трёхмерное перемещение, использующее однородные координаты, вектор перемещения (dx, dy, dz) вставляется в четвёртый столбец матрицы переноса 4×4:

(7)

используется в методе *glm::Translate(x,y,z)*.

1.2.2. Модель сферы

В компьютерной графике для описания трёхмерных объектов применяются различные типы моделей, однако на практике наиболее часто используется полигональная сетка [4], как мы можем увидеть на рисунке 1.

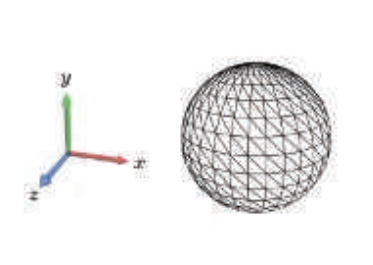


Рисунок 1 – Сфера из полигонов

Полигональные сетки особенно эффективны в приложениях реального времени, так как современные графические процессоры оптимизированы для их обработки. Наиболее универсальной формой полигона является треугольник, и треугольные сетки получили широкое распространение благодаря своей простоте и надёжности. Тем не менее, во многих задачах моделирования и визуализации предпочтение отдаётся четырёхугольным (квадратным) сеткам, так как они обеспечивают более регулярную структуру и облегчают последующую обработку (например, текстурирование или анимацию).

В данной работе используется четырёхугольная полигональная сетка, построенная с помощью режима рендеринга GL\_QUAD\_STRIP, который позволяет эффективно формировать полосы из последовательных четырёхугольников. Это позволяет минимизировать количество вершин и обеспечить связное построение поверхности.

Для замкнутых поверхностей без отверстий справедлива формула Эйлера:

v − e + f = 2, где v — число вершин, e — число рёбер, а f — число граней сетки. В треугольной сетке каждое ребро принадлежит двум граням, а каждая грань состоит из трёх рёбер. Следовательно, выполняется соотношение: 2e = 3f.

Для четырёхугольной сетки аналогично можно вывести: 2e = 4f, так как каждая грань состоит из четырёх рёбер, и каждое ребро также принадлежит двум гранямх[4].

1.3. Реализация движения маятника

Колебания, предоставленные самим себе, называют свободными. Они совершаются только за счет первоначально заданной энергии, без дальнейшего воздействия на систему колебаний.

Если система консервативная, то при колебаниях не происходит изменения энергии системы. Такие колебания являются незатухающими. Уравнение свободных незатухающих колебаний, или уравнение гармонического осциллятора, имеет вид:

(8)

где (м/с²) – ускорение, 𝑥(м) – отклонение от положения равновесия, \_\_\_\_ (рад/с) – собственная циклическая частота колебательной системы.

Решением данного линейного дифференциального однородного уравнения является периодическая функция:

(9)

Свободные незатухающие колебания происходят с собственной частотой , которая определяется параметрами колебательной системы. В любой реальной колебательной системе есть силы сопротивления, действие которых приводит к тому, что энергия системы постепенно расходуется на нагревание окружающей среды и самой системы. В результате амплитуда колебаний со временем уменьшается. Такие свободные колебания называют затухающими. Во многих практических случаях силу механического сопротивления можно считать пропорциональной скорости:

(10)

или для проекции:

(11)

где (кг/с) – коэффициент сопротивления. С учетом сопротивления (8) принимает вид:

(12)

где (1/с) – коэффициент затухания, характеризующий быстроту затухания колебаний.

Аналитическим решением этого уравнения будет негармоническая функция:

(13)

где (м) – начальная амплитуда (максимальное смещение), − частота затухающих колебаний, (рад) – начальная фаза. Амплитуда затухающих колебаний , как видно, монотонно убывает со временем, и в этом смысле затухающие колебания нельзя назвать периодическими, т.к. состояние системы никогда в точности не повторяется.

. Описание разработанной программы

2.1. Освещение

В данной работе я использую модель освещения Ламберта вместе с использованием прожектора. В этой модели используется три вида освещения:

1) Окружающий свет

2) Диффузный свет, который поступает из прожектора.

3) Зеркальный свет, который создает блик на объекте.

2.2. Настройка сцены

Метод GL\_OpenGLInitialized выполняет первичную настройку графической сцены OpenGL при инициализации элемента OpenGLControl. Он отвечает за установку параметров отображения, проекционной и модельно-видовой матриц, а также включает базовые механизмы визуализации, такие как освещение и тест глубины.

Область вывода *gl.Viewport(0, 0, W, H)* задаёт часть окна, в которую будет производиться рендеринг. Она определяется размерами элемента управления OpenGL(W, H). Это необходимо для корректного отображения сцены независимо от размеров окна. Далее устанавливается перспективная проекция. Она позволяет передавать ощущение глубины, за счёт сужения объектов вдали. Параметры метода Perspective: угол обзора (45°), соотношение сторон (ширина/высота), передняя и задняя плоскости отсечения (1.0 и 2000.0 единиц):

*gl.Viewport(0, 0, W, H);*

*gl.MatrixMode(OpenGL.GL\_PROJECTION);*

*gl.LoadIdentity();*

*gl.Perspective(45.0f, (float)W / H, 1.0f, 2000.0f);*

Настройка модельно-видовой матрицы, производится с помощью функции LookAt, она определяет положение виртуальной камеры: первый вектор (distanceX, distanceY, distanceZ) – положение камеры в мировом пространстве, второй вектор (0, 0, 0) – точка, на которую направлена камера, третий вектор  
(0, 1, 0) – вектор «вверх» (ось Y). Это определяет ориентацию камеры и формирует основу для построения сцены:

*gl.MatrixMode(OpenGL.GL\_MODELVIEW);*

*gl.LoadIdentity();*

*gl.LookAt(distanceX, distanceY, distanceZ,*

*0, 0, 0,*

*0, 1, 0);*

Тест глубины позволяет корректно отображать взаимное расположение объектов в 3D-пространстве. Он обеспечивает, чтобы пиксели, расположенные ближе к камере, перекрывают те, что находятся дальше:

*gl.Enable(OpenGL.GL\_DEPTH\_TEST);*

2.3. Цикл отрисовки

Метод GL\_OpenGLDraw вызывается при каждом кадре рендеринга и отвечает за отрисовку содержимого 3D-сцены в окне. Он реализует цикл обновления изображения, выполняемый при включённой анимации или при необходимости перерисовки.

Устанавливается голубой цвет фона, затем очищаются: цветовой буфер (GL\_COLOR\_BUFFER\_BIT) – для перерисовки сцены, буфер глубины (GL\_ DEPTH\_BUFFER\_BIT) – для корректной работы теста глубины:

*gl.ClearColor(0.53f, 0.81f, 0.92f, 1.0f);*

*gl.Clear(OpenGL.GL\_COLOR\_BUFFER\_BIT | OpenGL.GL\_DEPTH\_BUFFER\_BIT);*

Далее устанавливается модельно-видовая матрица и камера, после вызываются методы отрисовки моделей.

2.3. Освещение

Метод SetupLighting выполняет настройку основного освещения сцены в OpenGL.

Включение системы освещения: GL\_LIGHTING – включает глобальную систему освещения в OpenGL, GL\_LIGHT0 – активирует один из доступных источников света:

*gl.Enable(OpenGL.GL\_LIGHTING);*

*gl.Enable(OpenGL.GL\_LIGHT0);*

Использование материала по цвету объекта происходит с помощью следующих методов: GL\_COLOR\_MATERIAL позволяет использовать текущий цвет, как материал для освещения, параметр GL\_AMBIENT\_AND\_DIFFUSE означает, что указанный цвет влияет одновременно на фоновое и рассеянное освещение, GL\_FRONT\_AND\_BACK – применяет свойства материала к передним и задним граням объекта. Это позволяет не задавать отдельно материал, а управлять внешним видом через цвет объекта:

*gl.Enable(OpenGL.GL\_COLOR\_MATERIAL);*

*gl.ColorMaterial(OpenGL.GL\_FRONT\_AND\_BACK, OpenGL.GL\_AMBIENT\_AND\_DIFFUSE);*

Каждый источник света может включать в себя три основных компонента:

1) **Ambient (окружающий свет):**

* Симулирует мягкое, рассеянное освещение от всех направлений.
* Даже если объект не направлен на свет, он будет слабо освещён.
* Здесь задается значение 0.1.

2) **Diffuse (рассеянный свет):**

* Освещает поверхности, ориентированные к источнику света.
* Зависит от угла между направлением к свету и нормалью поверхности.
* Основной эффект объема и формы. Значения около 0.8-0.9 дают хороший рассеянный свет.

3) **Specular (зеркальное отражение):**

* Симулирует блики на гладких, глянцевых поверхностях.
* Зависит от угла отражения и положения камеры.
* Здесь используется значение 0.4, дающее умеренный блеск.

Код, реализующий источники света можно увидеть ниже:

*gl.Light(LightName.Light0, LightParameter.Ambient, new float[] { 0.1f, 0.1f, 0.1f, 1 });*

*gl.Light(LightName.Light0, LightParameter.Diffuse, new float[] { 0.8f, 0.8f, 0.9f, 1 });*

*gl.Light(LightName.Light0, LightParameter.Specular, new float[] { 0.4f, 0.4f, 0.4f, 1 });*

2.4. построение сферы

Визуализация сферы в OpenGL может быть выполнена различными способами. Один из наиболее понятных и наглядных – разбиение сферы на полосы широты и долготы, формирующие сетку из четырёхугольников. В данной реализации используется функция DrawSphere, которая строит сферу с заданным радиусом и числом сегментов по широте и долготе.

**2.4.1. Полярные координаты**

В полярных координатах любая точка на поверхности единичной сферы может быть определена по формулам:

В коде:

* *lat0* и *lat1* – два соседних значения широты (от -π/2 до +π/2).
* *lng* – значение долготы (от 0 до 2π).
* *zr0 = radius \* cos(lat0)* – проекция текущей широты на XY-плоскость.
* *z0 = radius \* sin(lat0)* – координата по оси Z.

**2.4.2. Сеточная аппроксимация сферы**

Визуально сфера разбивается на горизонтальные полосы, каждая из которых состоит из сегментов, соединяющих два соседних уровня широты (*lat0, lat1*) с помощью вертикальных полос по долготе (*lonSegments*). Это формирует четырёхугольную сетку, отрисованную с помощью режима GL\_QUAD\_STRIP.

Для каждого сегмента:

* Строится лента из четырёхугольников между *lat0* и *lat1*.
* Каждая полоса создаётся внутри цикла *for*, проходящего по долготе.

Таким образом, сетка состоит из *latSegments × lonSegments* четырёхугольников, визуально образующих сферу.

**3. Нормали и текстурные координаты**

Нормаль каждой вершины (gl.Normal) совпадает с направлением радиуса из центра сферы.

Текстурные координаты gl.TexCoord(u, v) задаются на основе параметров (u, v), соответствующих нормализованным долготе и широте.

**4. Использование GL\_QUAD\_STRIP**

Режим GL\_QUAD\_STRIP строит полосу из соединённых четырёхугольников. Каждая пара вершин, добавляемая в цикле, образует один новый четырёхугольник вместе с предыдущей парой. Это позволяет минимизировать количество операций и вершин при построении регулярной сетки.

Метод DrawSphere создаёт реалистичную сферическую модель с четырёхугольной топологией. Он представляет собой пример процедурной генерации геометрии с использованием тригонометрических преобразований и параметризации поверхности.

3.2. Интерфейс программы

Использование клавиш управления точками проекции центра движущегося шара, показывающих движение маятника можно увидеть на рисунках 5-6.

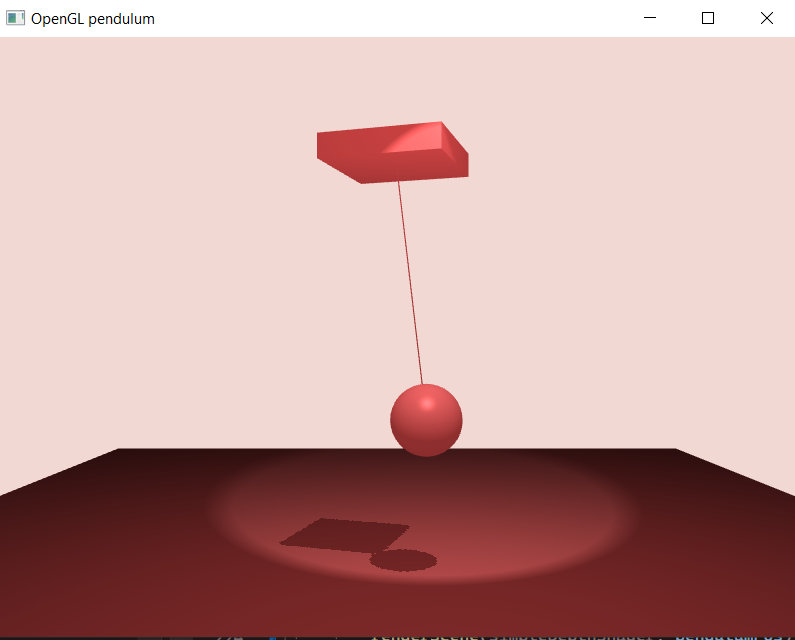


Рисунок 5 – Маятник, при выключении точек (клавиша Р)

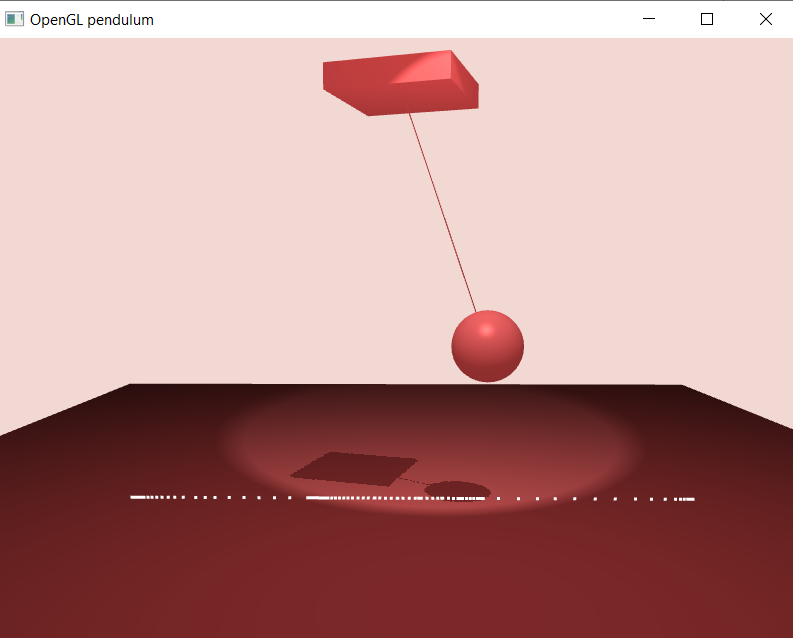


Рисунок 6 – Маятник, при включении точек (клавиша O)

Использовании клавиш управления камерой можно увидеть на рисунках 7-9.



Рисунок 7 – Маятник (вид сбоку)

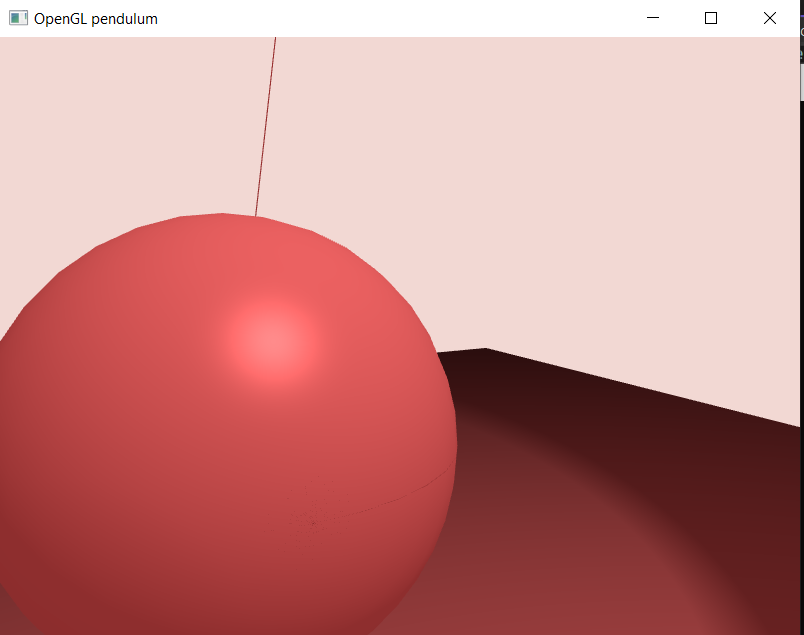


Рисунок 8 – Маятник вблизи

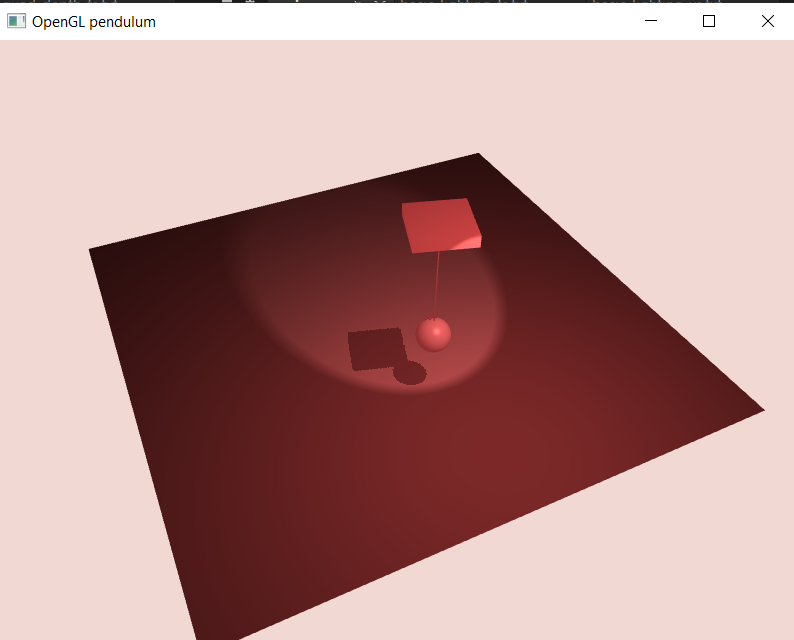


Рисунок 9 – Маятник (вид сверху)

3. Примеры работы программы

3.1. Тест №1

Маятник начинает свое движение с раскачивания из стороны в сторону. В модели используется разное освещение, это можно увидеть на рисунках 10-17. В данном тесте используются следующие параметры: длинна нити – 2.5, коэффициент сопротивления – 0.05, масса шара – 1.0, угол отклонения – 90°.

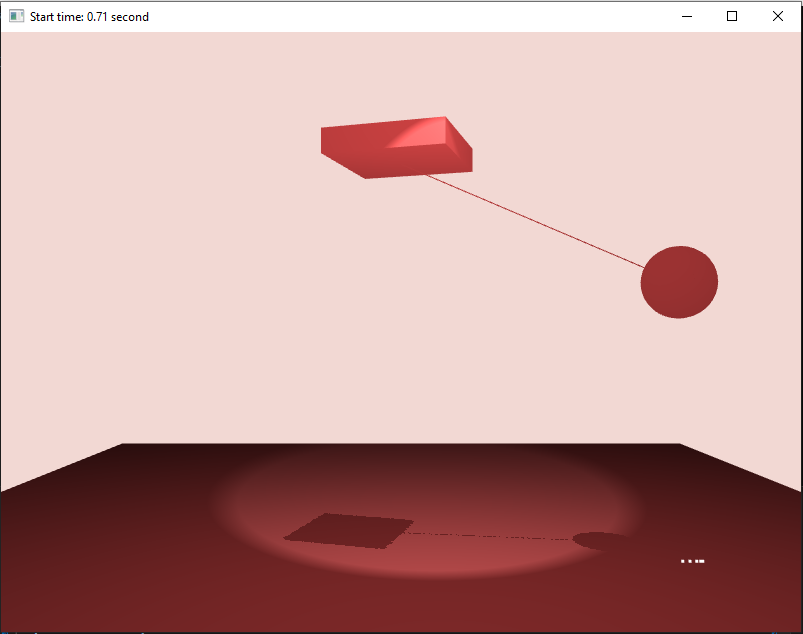


Рисунок 10 – Маятник при t = 0.71 секунда, угол 67.25°

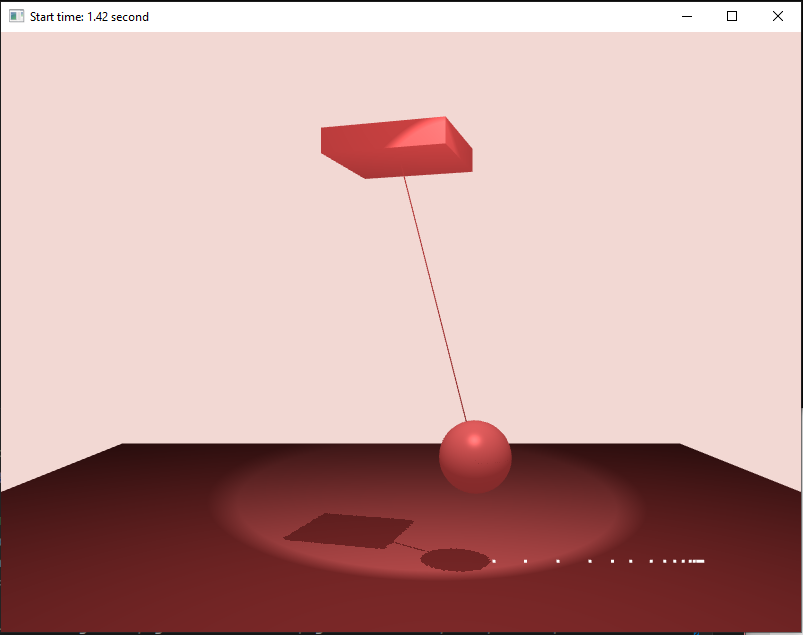


Рисунок 11 – Маятник при t = 1.42 секунды, угол 14.12°

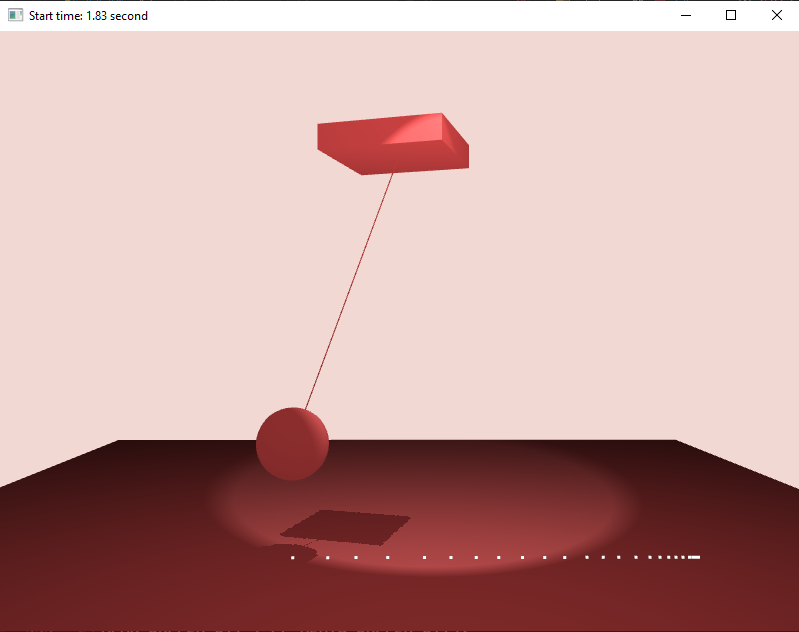


Рисунок 12 – Маятник при t = 1.83 секунды, угол 20.78° влево

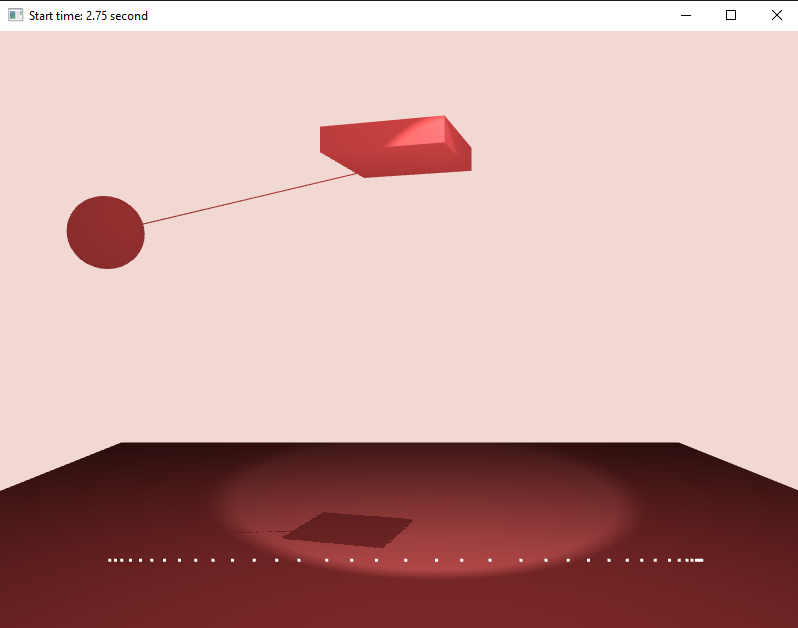


Рисунок 13 – Маятник при t = 2.75 секунд, угол 76.82° влево

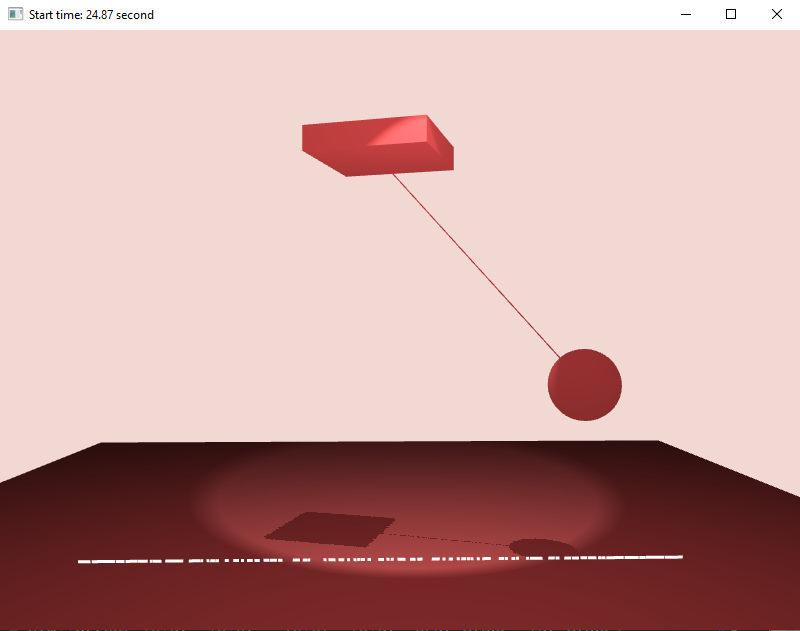


Рисунок 14 – Маятник при t = 24.87 секунд, угол 42.42°

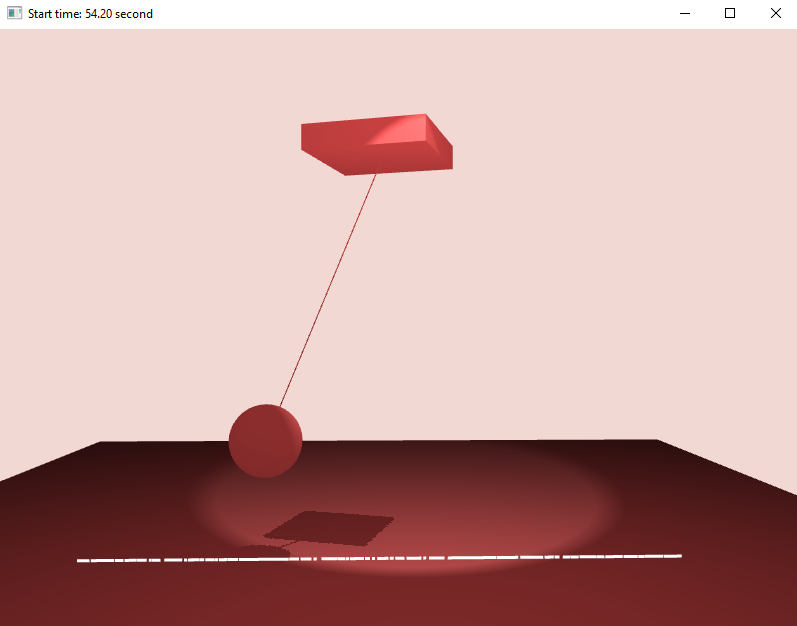


Рисунок 15 – Маятник при t = 54.20 секунды, угол 22.31° влево

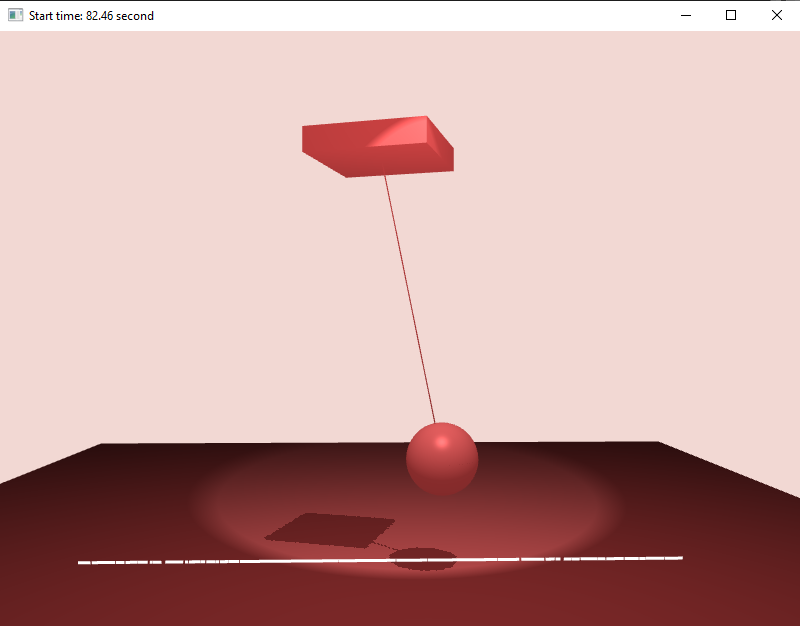


Рисунок 16 – Маятник при t = 82.46 секунд, угол 11.45°

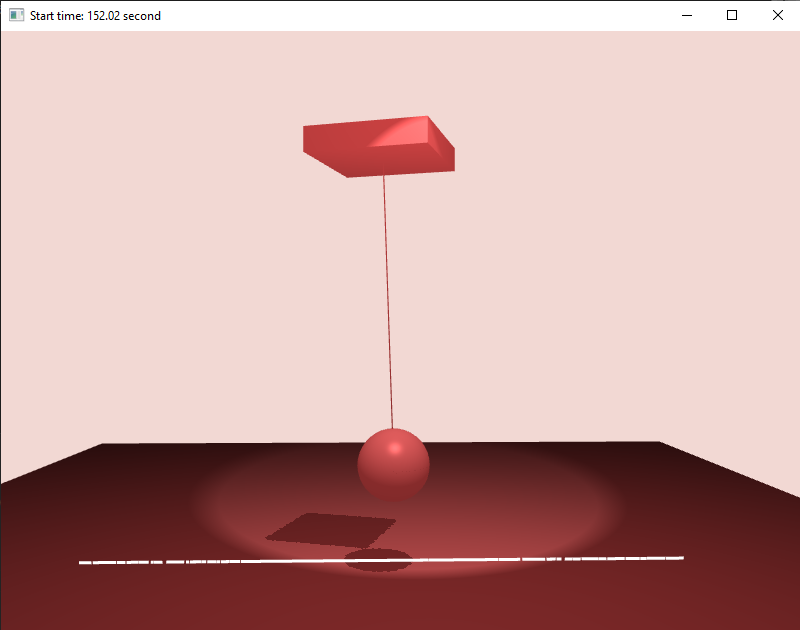


Рисунок 17 – Маятник при t = 152.02 секунд, угол 1.96°

3.2. Тест №2

В данном тесте используются следующие параметры: длинна нити – 1.5, коэффициент сопротивления – 0.05, масса шара – 1.0, угол отклонения – 90°. Результат работы программы можно увидеть на рисунках 18-23.

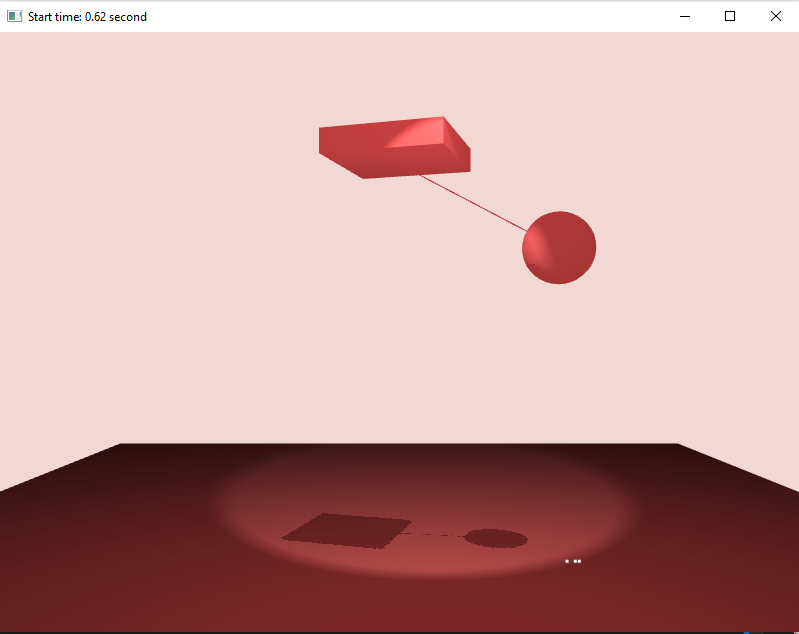


Рисунок 18 – Маятник при t = 0.62 секунды, угол 61.55°

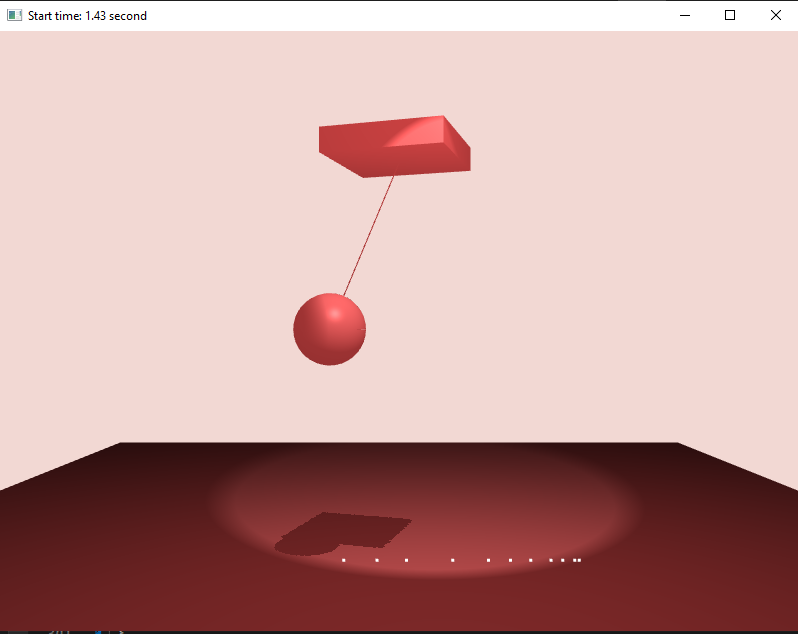


Рисунок 19 – Маятник при t = 1.43 секунды, угол 22.18° влево

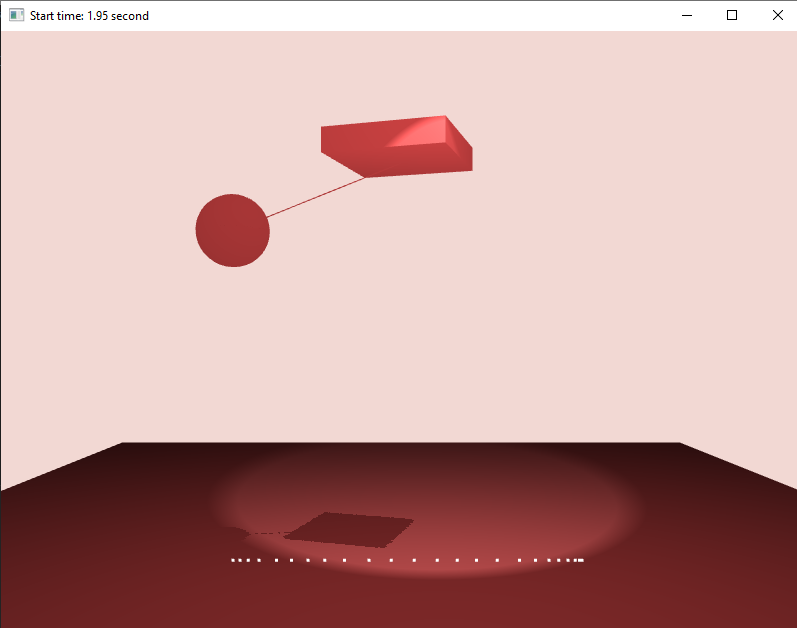


Рисунок 20 – Маятник при t = 1.95 секунд, угол 68.34° влево

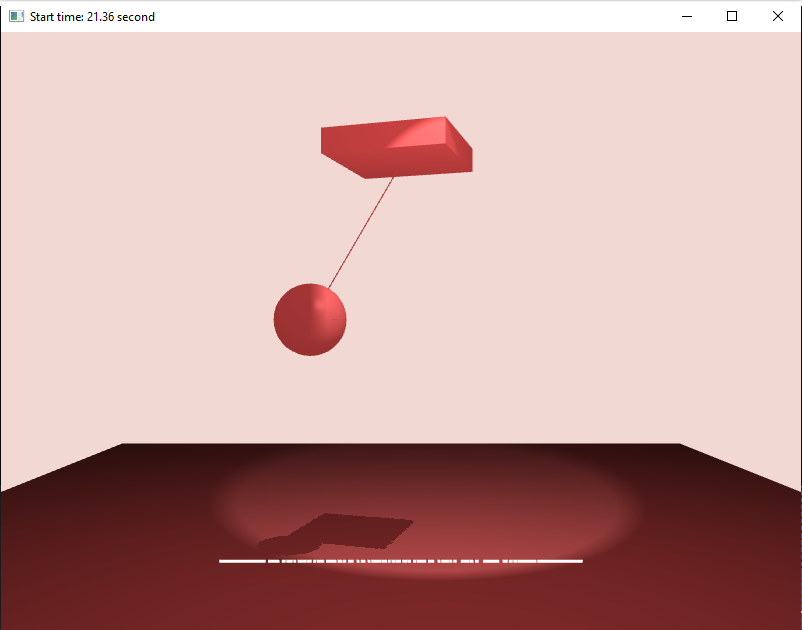


Рисунок 21 – Маятник при t = 21.36 секунд, угол 30.32° влево

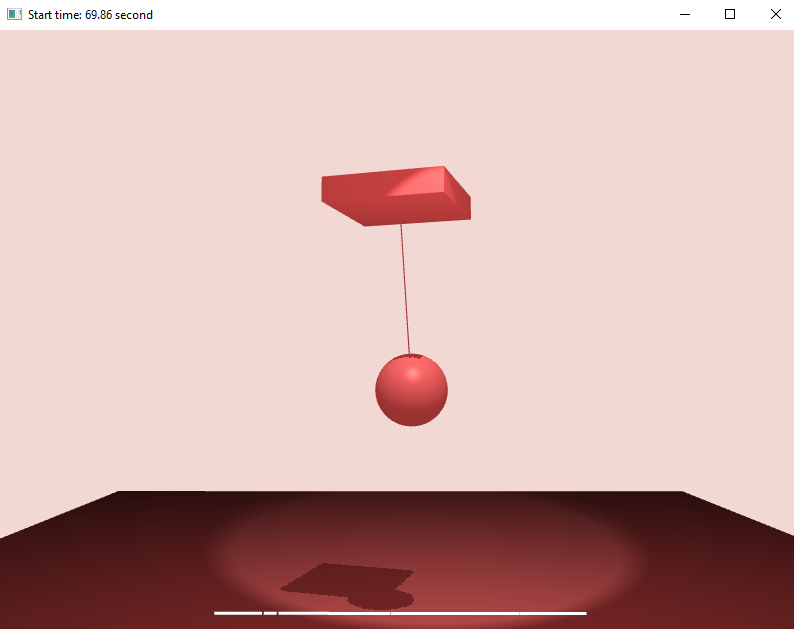


Рисунок 22 – Маятник при t = 69.86 секунд, угол 3.14°

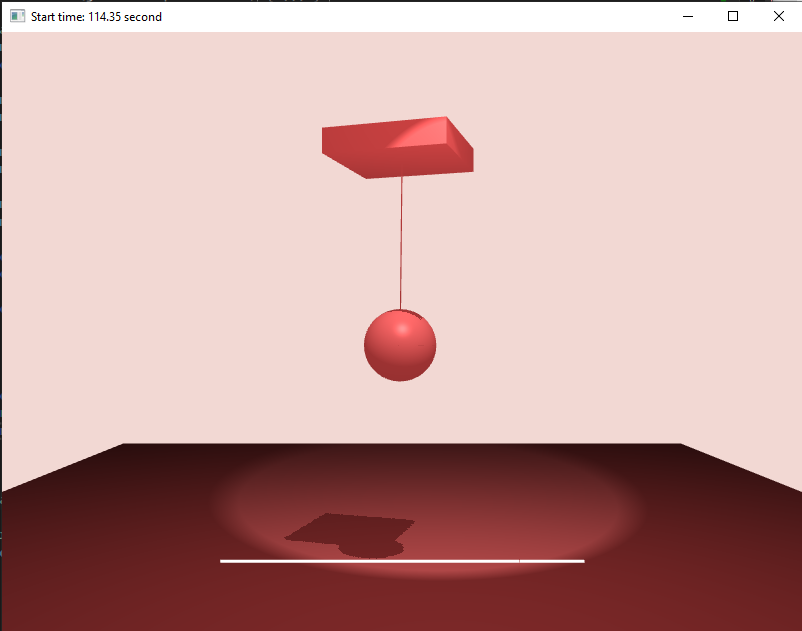


Рисунок 23 – Маятник при t = 114.35 секунд, угол 0.73°

3.3. Тест №3

В данном тесте используются следующие параметры: длинна нити – 2.5, коэффициент сопротивления – 0.1, масса шара – 1.0, угол отклонения – 90°. Результат работы программы можно увидеть на рисунках 24-28.

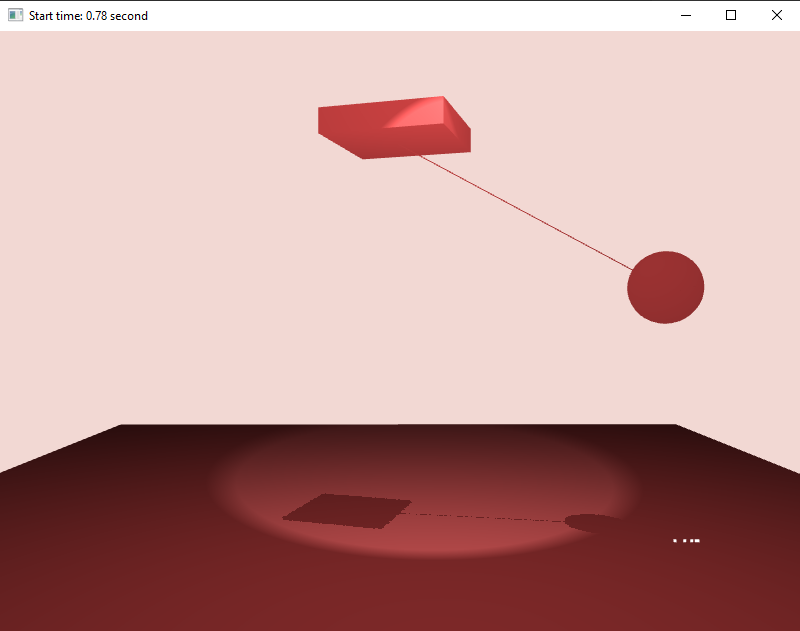


Рисунок 24 – Маятник при t = 0.78 секунды, угол 61.88°

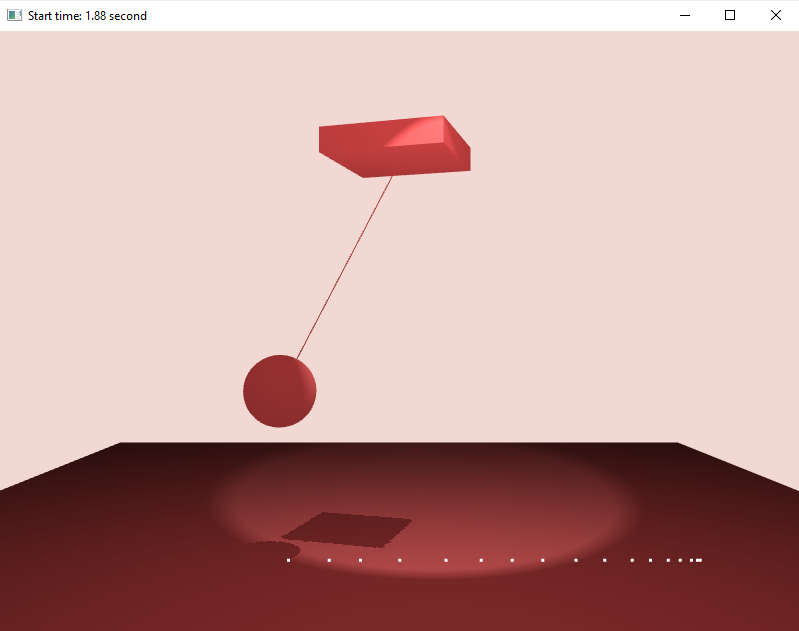


Рисунок 25 – Маятник при t = 1.88 секунд, угол 23.74° влево

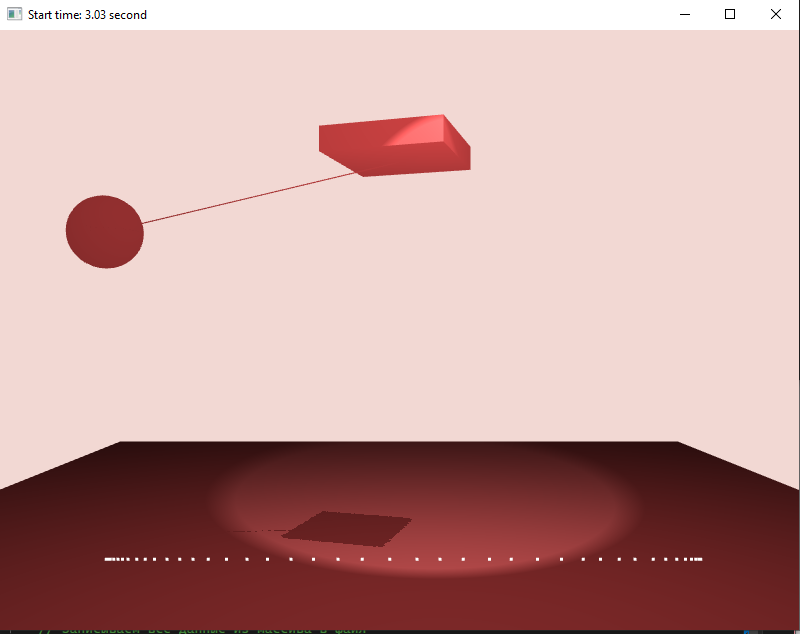


Рисунок 26 – Маятник при t = 3.03 секунд, угол 76.59° влево

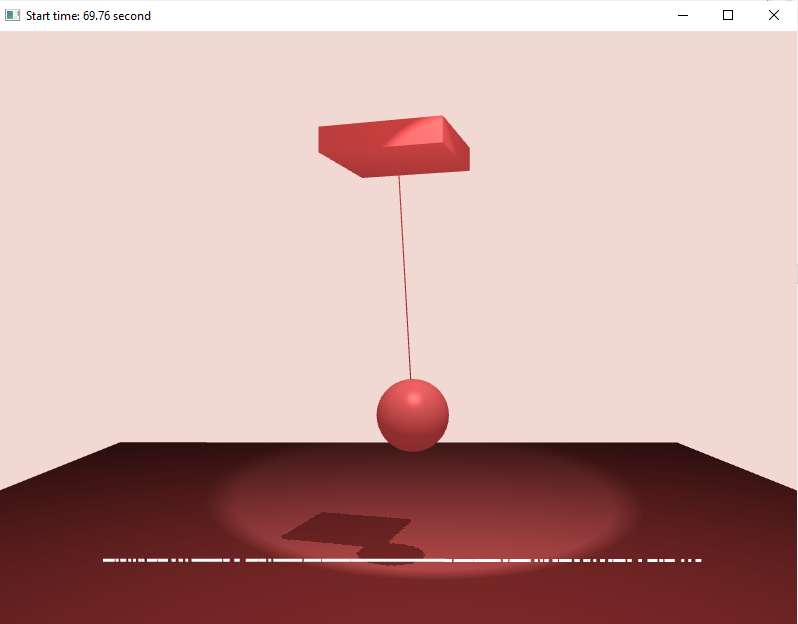


Рисунок 27 – Маятник при t = 69.76 секунд, угол 2.74°

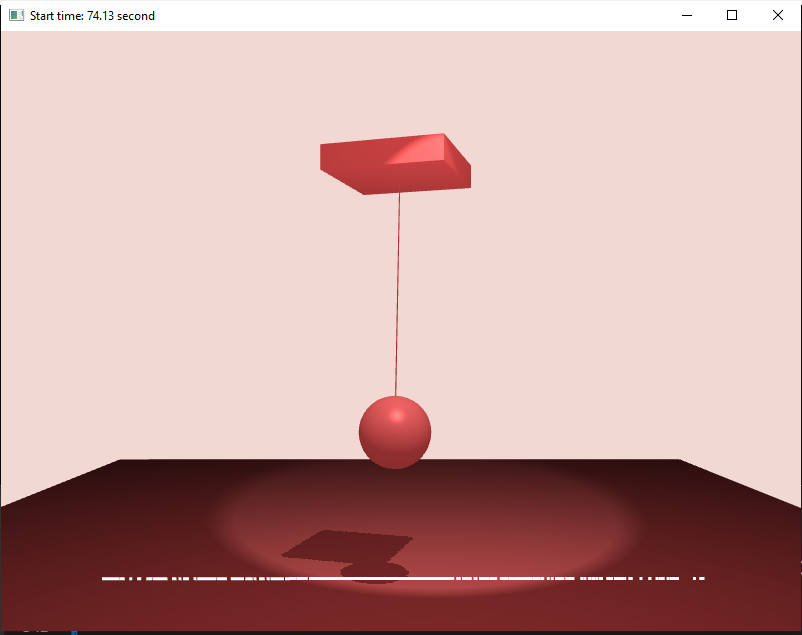


Рисунок 28 – Маятник при t = 74.13 секунд, угол 0.85° влево

3.4. Тест №4

В данном тесте используются следующие параметры: длинна нити – 2.5, коэффициент сопротивления – 0.05, масса шара – 5.0, угол отклонения – 90°. Результат работы программы можно увидеть на рисунках 29-34.

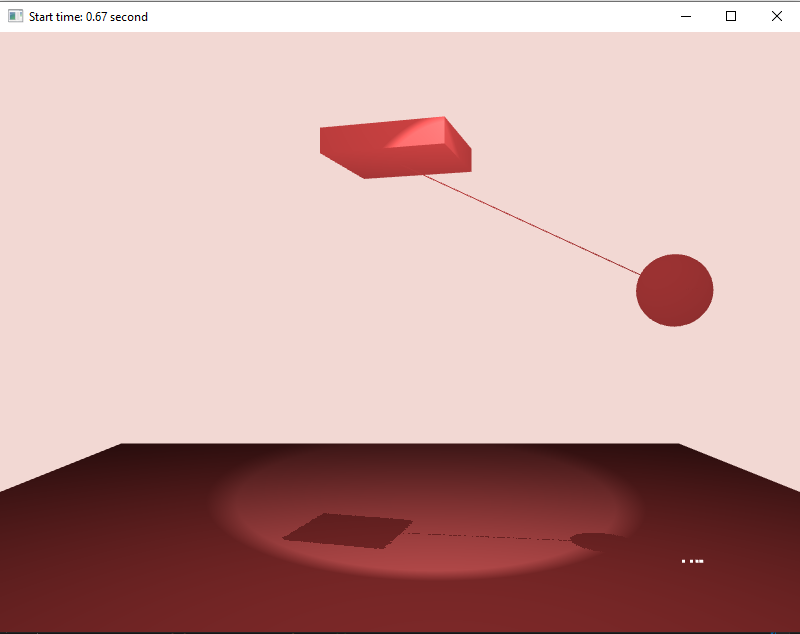


Рисунок 29 – Маятник при t = 0.67 секунд, угол 65.01°

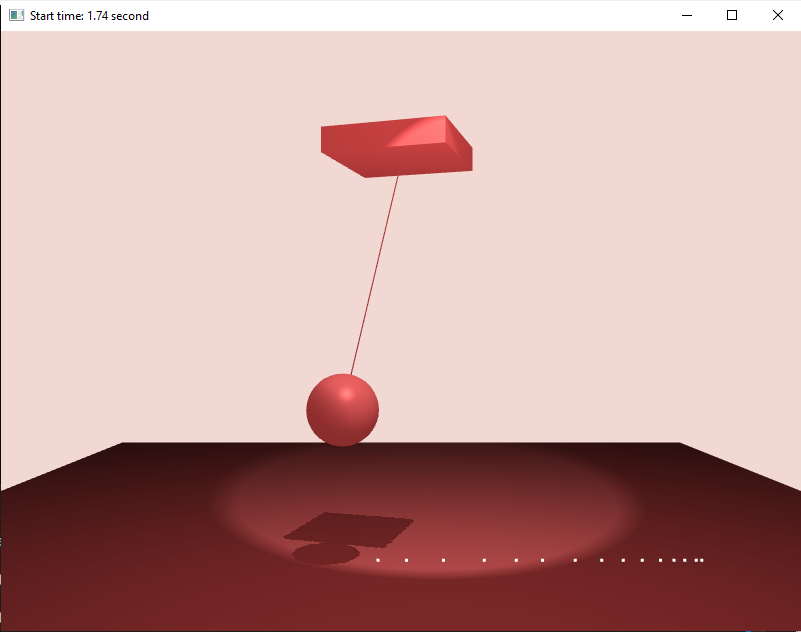


Рисунок 30 – Маятник при t = 1.74 секунды, угол 11.07° влево

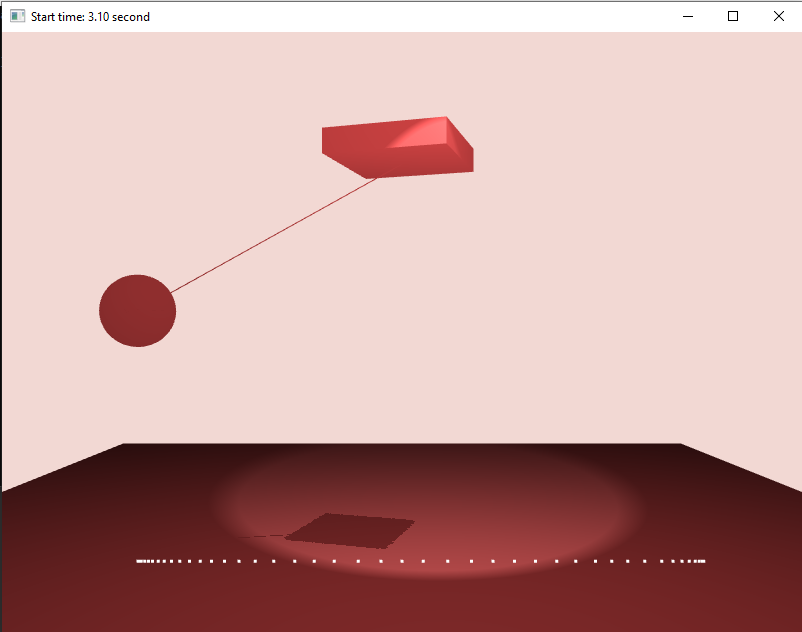


Рисунок 31 – Маятник при t = 3.10 секунд, угол 60.92° влево

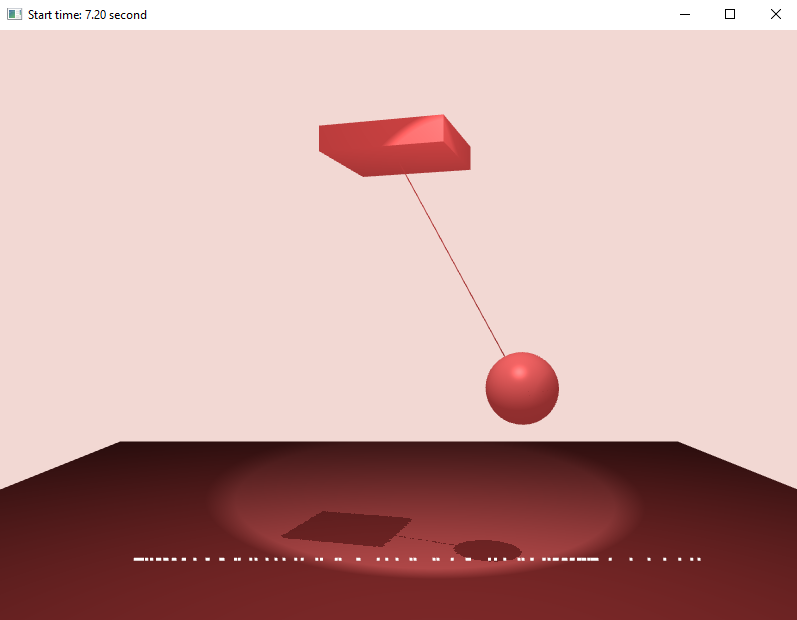


Рисунок 32 – Маятник при t = 7.20 секунд, угол 24.19°

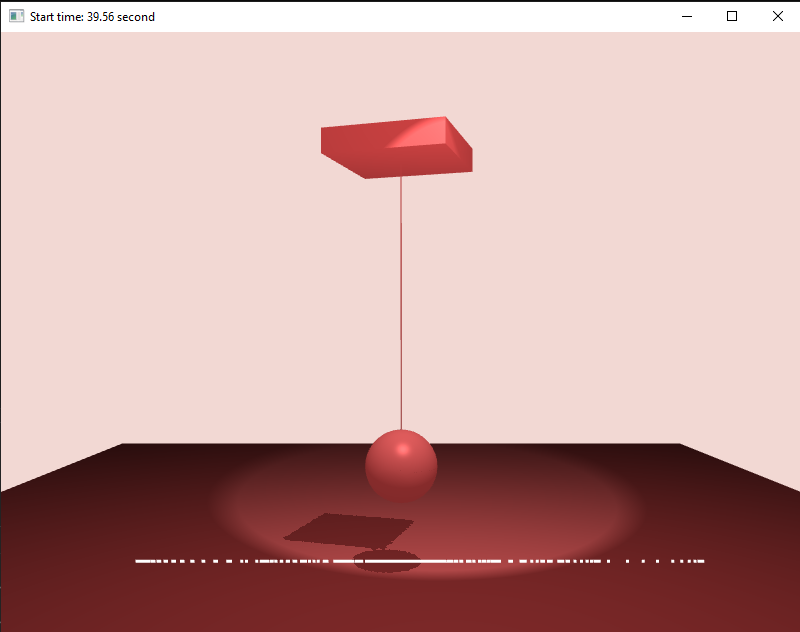


Рисунок 33 – Маятник при t = 39.56 секунд, угол 0.04°

3.5. Тест №5

В данном тесте используются следующие параметры: длинна нити – 2.5, коэффициент сопротивления – 0.05, масса шара – 1.0, угол отклонения – 45°. Результат работы программы можно увидеть на рисунках 34-38.

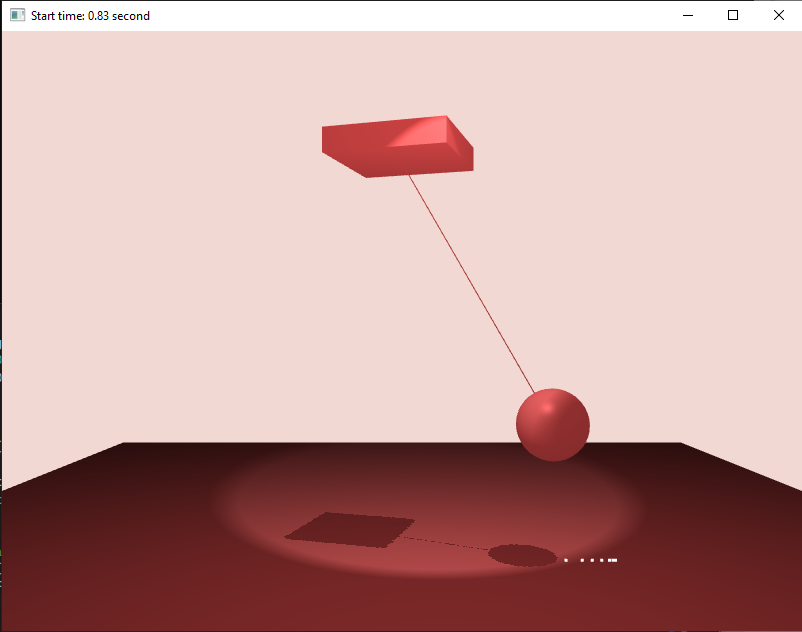


Рисунок 34 – Маятник при t = 0.83 секунды, угол 29.99°

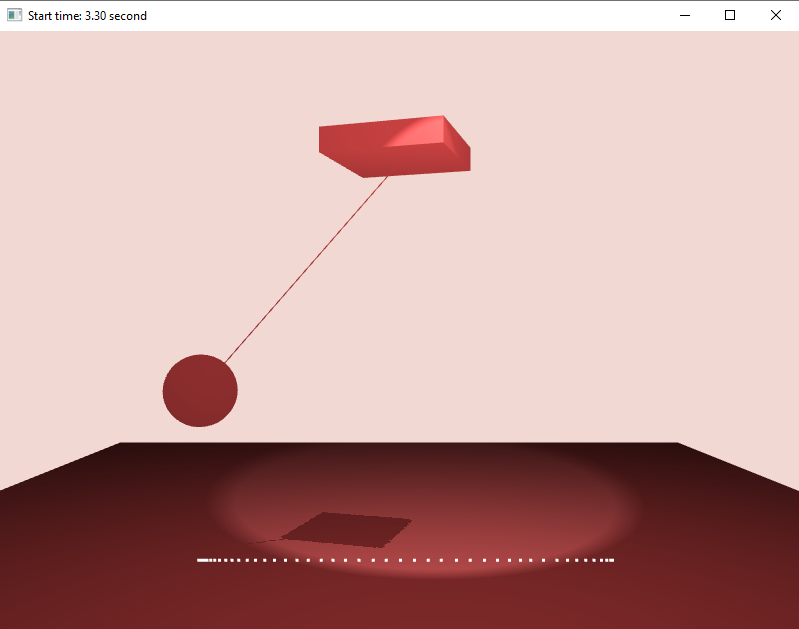


Рисунок 35 – Маятник при t = 3.30 секунд, угол 41.08° влево

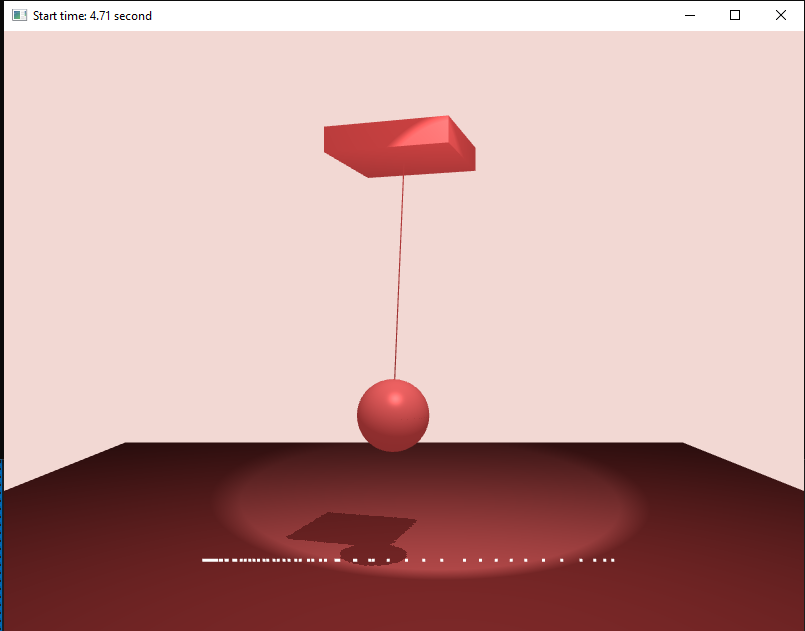


Рисунок 36 – Маятник при t = 4.71 секунда, угол 1.82° влево

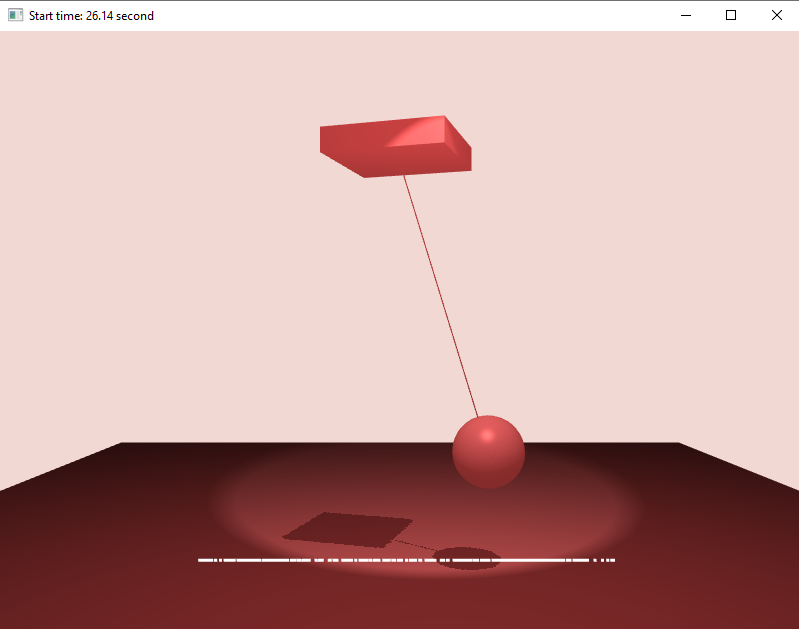


Рисунок 37 – Маятник при t = 26.14 секунд, угол 16.97°

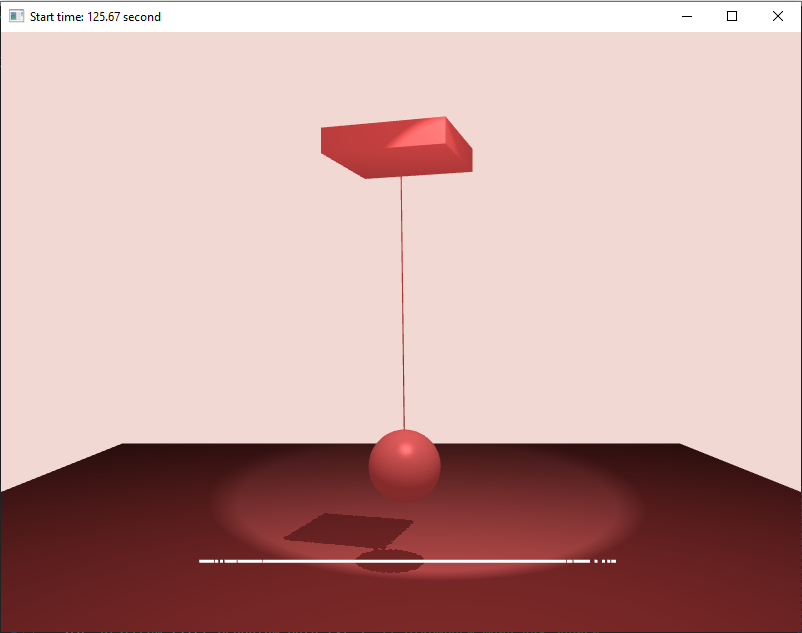


Рисунок 38 – Маятник при t = 125.67 секунд, угол 0.74°

Плавное раскачивание маятника успешно протестировано: движение маятника происходит без рывков, с правильным изменением скорости и направления, что подтверждает корректность физической модели.

В зависимости от угла света прожектора меняется освещение маятника: при попадании шара маятника в область его действия он становится ярко освещённым, и этот эффект исчезает за пределами зоны освещения, что подтверждает корректность освещения модели.

Также в ходе проведённых пяти тестов была проанализирована зависимость времени работы маятника от различных параметров. Полученные результаты позволяют сделать следующие выводы:

• Длина нити напрямую влияет на продолжительность колебаний: при уменьшении длины с 2,5 до 1,5 единиц время остановки маятника сократилось с ~170 до ~120 секунд. Это происходит, потому что уменьшается период и маятник быстрее теряет энергию на сопротивления.

• Коэффициент сопротивления среды оказывает значительное влияние на затухание колебаний. При увеличении сопротивления с 0,05 до 0,1 время остановки сократилось до ~80 секунд.

• Увеличение массы маятника (с 1 до 5 единиц) также привело к более быстрому затуханию – маятник остановился через ~40 секунды. Это происходит, потому что инерция увеличивается, а сопротивление возрастает.

• Уменьшение начального угла отклонения с 90° до 45° сокращает запас начальной энергии, что приводит к более быстрому затуханию – в данном случае маятник остановился через ~130 секунд.

Все тесты подтверждают, что увеличение сопротивления среды, массы маятника или уменьшение длины и начального отклонения приводят к более быстрому завершению колебаний.

Заключение

Разработанная визуализация физической симуляции маятника с использованием технологии OpenGL представляет собой увлекательный и наглядный способ продемонстрировать основные принципы механики и физики. В этой симуляции зрителю предоставляется возможность наблюдать за действием таких фундаментальных явлений, как сила тяжести, угловое ускорение, а также эффект затухания, что способствует более глубокому пониманию сложных физических процессов.

Реализованная модель маятника воспроизводит его движение в режиме реального времени, предлагая зрителям реалистичное и детализированное отображение динамики системы. Благодаря высокой точности вычислений и визуализации, симуляция не только передает характерное поведение маятника, но и позволяет учитывать влияние различных параметров, таких как длина нити, начальный угол отклонения и коэффициент затухания.

Эта визуализация становится особенно полезной в образовательных целях, позволяя студентам и любителям физики с легкостью освоить сложные концепции через практическое наблюдение. Одновременно она может служить основой для дальнейшего углубления в области численного моделирования, а также разработки аналогичных симуляций для других физических систем.

Список литературы

1. Документация по стандартной библиотеке OpenGL. [Электронный ресурс] URL: https://learn.microsoft.com/ ru-ru/windows/win32/opengl

/opengl (дата обращения: 15.04.2025).

1. Документация по WinAPI. [Электронный ресурс] URL: https://learn

.microsoft.com/ru-ru/windows/win32/learnwin32/learn-to-program-for-windows (дата обращения: 11.04.2025).

1. De Vries Joey. Learn OpenGL Graphics Programming in a step-by-step fashion (дата обращения: 08.04.2025).
2. Han JungHyun. Introduction to Computer Graphics with OpenGL ES (дата обращения: 09.04.2025).

Приложение. Код программы

#include <glad/glad.h>

#include <GLFW/glfw3.h>

#include <iostream>

#include <sheder\_s.h>

#define STB\_IMAGE\_IMPLEMENTATION

#include <stb\_image.h>

#include <stdlib.h>

#include <array>

#define \_STDCALL\_SUPPORTED

#define \_M\_IX86

#include "../../include/glut.h"

#define \_USE\_MATH\_DEFINES

#include <cmath>

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#include <glm/gtc/type\_ptr.hpp>

#include "../../include/camera.h"

struct Vec2 {

float x, y;

};

struct Vec3 {

float x, y, z;

};

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos);

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset);

void processInput(GLFWwindow\* window, Shader shader, Shader debugDepthQuad,

Shader simpleDepthShader, Shader shaderPoint);

unsigned int loadTexture(const char\* path);

void renderScene(const Shader& shader, glm::vec3 pendulumPos);

void drawingPoints(float t, glm::vec3 pendulumPos);

void renderPlatform();

void renderSphere();

void renderLine(glm::vec3 pendulumPos);

void renderPoint(glm::vec3 pendulumPos);

Vec2 getPendulumPosition(float t);

Vec3 display(float timeElapsed);

std::vector<float> generateSphereVertices(float radius, int sectorCount, int stackCount);

std::vector<GLuint> generateSphereIndices(int sectorCount, int stackCount);

std::array<Shader, 4> createShaders();

void ckle(GLFWwindow\* window, Shader shader, Shader simpleDepthShader,

Shader shaderPoint, Shader debugDepthQuad);

// Константы

unsigned int SCR\_WIDTH = 800;

unsigned int SCR\_HEIGHT = 600;

const unsigned int SHADOW\_WIDTH = 1024, SHADOW\_HEIGHT = 1024;

unsigned int depthMapFBO;

unsigned int depthMap;

// Камера

Camera camera(glm::vec3(0.0f, 0.0f, 5.0f));

float lastX = (float)SCR\_WIDTH / 2.0;

float lastY = (float)SCR\_HEIGHT / 2.0;

bool firstMouse = true;

// Тайминги

float deltaTime = 0.0f;

float lastFrame = 0.0f;

// Параметры освещения

glm::vec3 lightPos(0.3f, 2.1f, 0.8f);

// Меши

//glm::vec3 pendulumPos(0.0f, -1.0f, -1.0f);

unsigned int planeVAO;

glm::vec3 Points[1000];

bool point = true;

int currentIndex = 0;

int counter = 0;

int main()

{

// glfw: инициализация и конфигурирование

glfwInit();

glfwWindowHint(GLFW\_SAMPLES, 4);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

// glfw: создание окна

GLFWwindow\* window = glfwCreateWindow(SCR\_WIDTH, SCR\_HEIGHT, "OpenGL pendulum", NULL, NULL);

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return -1;

}

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

glfwSetCursorPosCallback(window, mouse\_callback);

glfwSetScrollCallback(window, scroll\_callback);

// Сообщаем GLFW, чтобы он захватил наш курсор

//glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);

// glad: загрузка всех указателей на OpenGL-функции

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

{

std::cout << "Failed to initialize GLAD" << std::endl;

return -1;

}

// Конфигурирование глобального состояния OpenGL

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_MULTISAMPLE);

// Компилирование нашей шейдерной программы

/\* Shader shader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/shadow\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/shadow\_fs.txt");

// threadShader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/shadow\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/thread\_fs.txt");

Shader debugDepthQuad("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/quad\_depth\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/quad\_depth\_fs.txt");

Shader simpleDepthShader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/mapping\_depth\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/mapping\_depth\_fs.txt");

Shader shaderPoint("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/point\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/point\_fs.txt");

\*/

std::array<Shader, 4> shaders = createShaders();

// Указание вершинных данных и настройка вершинных атрибутов

float planeVertices[] = {

// координаты // нормали // текстурные координаты

5.0f, -2.0f, 2.0f, 0.0f, 1.0f, 0.0f, 25.0f, 0.0f,

-5.0f, -2.0f, 2.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f,

-5.0f, -2.0f, -8.0f, 0.0f, 1.0f, 0.0f, 0.0f, 25.0f,

5.0f, -2.0f, 2.0f, 0.0f, 1.0f, 0.0f, 25.0f, 0.0f,

-5.0f, -2.0f, -8.0f, 0.0f, 1.0f, 0.0f, 0.0f, 25.0f,

5.0f, -2.0f, -8.0f, 0.0f, 1.0f, 0.0f, 25.0f, 25.0f

};

// VAO пола

unsigned int planeVBO;

glGenVertexArrays(1, &planeVAO);

glGenBuffers(1, &planeVBO);

glBindVertexArray(planeVAO);

glBindBuffer(GL\_ARRAY\_BUFFER, planeVBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(planeVertices), planeVertices, GL\_STATIC\_DRAW);

//glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)0);

glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(3 \* sizeof(float)));

glEnableVertexAttribArray(1);

// glVertexAttribPointer(2, 2, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(6 \* sizeof(float)));

glBindVertexArray(0);

// Загрузка текстур

//unsigned int woodTexture = loadTexture("C:/Users/pm14g/dev/Glitter/textures/wood.png");

// Настраиваем карту глубины FBO

glGenFramebuffers(1, &depthMapFBO);

// Создаем текстуры глубины

glGenTextures(1, &depthMap);

glBindTexture(GL\_TEXTURE\_2D, depthMap);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT, SHADOW\_WIDTH, SHADOW\_HEIGHT, 0, GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

// Прикрепляем текстуру глубины в качестве буфера глубины для FBO

glBindFramebuffer(GL\_FRAMEBUFFER, depthMapFBO);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_TEXTURE\_2D, depthMap, 0);

glDrawBuffer(GL\_NONE);

glReadBuffer(GL\_NONE);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

// Конфигурация шейдеров

shaders[0].use();

//shader.setInt("diffuseTexture", 0);

shaders[0].setInt("shadowMap", 0);

shaders[1].use();

shaders[1].setInt("depthMap", 0);

// Цикл рендеринга

ckle(window, shaders[0], shaders[1], shaders[2], shaders[3]);

glDeleteVertexArrays(1, &planeVAO);

glDeleteBuffers(1, &planeVBO);

glfwTerminate();

return 0;

}

void ckle(GLFWwindow\* window, Shader shader, Shader debugDepthQuad,

Shader simpleDepthShader, Shader shaderPoint) {

while (!glfwWindowShouldClose(window))

{

float currentFrame = glfwGetTime();

deltaTime = currentFrame - lastFrame;

lastFrame = currentFrame;

glm::vec3 pendulumPos(0.0f, -1.0f, -1.0f);

Vec2 q = getPendulumPosition(currentFrame);

// Vec3 q = display(currentFrame);

pendulumPos.x = q.x;

pendulumPos.y = q.y;

// pendulumPos.z = q.z;

// Обработка ввода

processInput(window, shader, debugDepthQuad, simpleDepthShader, shaderPoint);

// Рендер

glClearColor(0.95f, 0.85f, 0.83f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// 1. Рендеринг глубины сцены в текстуру (вид - с позиции источника света)

glm::mat4 lightProjection, lightView;

glm::mat4 lightSpaceMatrix;

float near\_plane = 1.0f, far\_plane = 7.5f;

lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near\_plane, far\_plane);

lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));

lightSpaceMatrix = lightProjection \* lightView;

// Рендеринг сцены глазами источника света

simpleDepthShader.use();

simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

glViewport(0, 0, SHADOW\_WIDTH, SHADOW\_HEIGHT);

glBindFramebuffer(GL\_FRAMEBUFFER, depthMapFBO);

glClear(GL\_DEPTH\_BUFFER\_BIT);

//glActiveTexture(GL\_TEXTURE0);

//glBindTexture(GL\_TEXTURE\_2D, woodTexture);

renderScene(simpleDepthShader, pendulumPos);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

// Сброс настроек области просмотра

glViewport(0, 0, SCR\_WIDTH, SCR\_HEIGHT);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// 2. Рендерим сцену как обычно, но используем при этом сгенерированную карту глубины/тени

glViewport(0, 0, SCR\_WIDTH, SCR\_HEIGHT);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

shader.use();

glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR\_WIDTH / (float)SCR\_HEIGHT, 0.1f, 100.0f);

glm::mat4 view = camera.GetViewMatrix();

shader.setMat4("projection", projection);

shader.setMat4("view", view);

// Устанавливаем uniform-переменные освещения

shader.setVec3("viewPos", camera.Position);

shader.setVec3("lightPos", lightPos);

shader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

shader.setVec3("light.direction", glm::vec3(0.0f, -1.0f, -1.0f));

shader.setFloat("light.cutOff", glm::cos(glm::radians(20.5f)));

shader.setFloat("light.outerCutOff", glm::cos(glm::radians(23.5f)));

shader.setVec3("light.position", lightPos);

shader.setFloat("light.constant", 1.0f);

shader.setFloat("light.linear", 0.09f);

shader.setFloat("light.quadratic", 0.032f);

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, depthMap);

renderScene(shader, pendulumPos);

/\* threadShader.use();

threadShader.setMat4("projection", projection);

threadShader.setMat4("view", view);

threadShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

threadShader.setMat4("model", glm::mat4(1.0f));

threadShader.setVec3("material.ambient", glm::vec3(0.5f, 1.0f, 1.0f));

threadShader.setVec3("material.diffuse", glm::vec3(0.5f, 1.0f, 1.0f));

threadShader.setVec3("normal", glm::vec3(0.0f, 1.0f, 0.0f));

threadShader.setVec3("light.position", lightPos);

threadShader.setVec3("light.direction", glm::vec3(0.0f, -1.0f, -1.0f));

threadShader.setFloat("light.cutOff", glm::cos(glm::radians(20.5f)));

threadShader.setFloat("light.outerCutOff", glm::cos(glm::radians(23.5f)));

threadShader.setVec3("light.ambient", glm::vec3(0.2f, 0.2f, 0.2f));

threadShader.setVec3("light.diffuse", glm::vec3(0.5f, 0.5f, 0.5f));

threadShader.setFloat("light.constant", 1.0f);

threadShader.setFloat("light.linear", 0.09f);

threadShader.setFloat("light.quadratic", 0.032f);

renderLine(pendulumPos);\*/

// Рендеринг на плоскости карты глубины для наглядной отладки

debugDepthQuad.use();

debugDepthQuad.setFloat("near\_plane", near\_plane);

debugDepthQuad.setFloat("far\_plane", far\_plane);

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, depthMap);

shaderPoint.use();

shaderPoint.setMat4("projection", projection);

shaderPoint.setMat4("view", view);

shaderPoint.setMat4("model", glm::mat4(1.0f));

if (point)

drawingPoints(currentFrame, pendulumPos);

counter++;

glfwSwapBuffers(window);

glfwPollEvents();

}

}

// Рендеринг 3D сцены

void renderScene(const Shader& shader, glm::vec3 pendulumPos)

{

// Пол

glm::mat4 model = glm::mat4(1.0f);

shader.setMat4("model", model);

glBindVertexArray(planeVAO);

glDrawArrays(GL\_TRIANGLES, 0, 6);

renderLine(pendulumPos);

model = glm::mat4(1.0f);

model = glm::translate(model, pendulumPos);

shader.setMat4("model", model);

renderSphere();

model = glm::mat4(1.0f);

model = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 1.5f, -1.0f));

model = glm::rotate(model, -0.3f, glm::vec3(0.0f, 1.0f, 0.0f));

shader.setMat4("model", model);

renderPlatform();

}

void drawingPoints(float t, glm::vec3 pendulumPos) {

if (currentIndex < sizeof(Points)) {

if (counter % 25 == 0) {

glm::vec3 Point(pendulumPos.x, -1.9f, pendulumPos.z);

Points[currentIndex] = Point;

currentIndex++;

}

for (int i = 0; i < currentIndex; ++i) {

renderPoint(Points[i]);

}

}

}

Vec2 getPendulumPosition(float t) {

float L = 2.5f;

float g = 9.81f;

float damping = 0.05f;

float timeScale = 0.5f;

float theta0 = 80.0f \* 3.14f / 180.0f;

float omega = std::sqrt(g / L);

float theta\_t = theta0 \* std::exp(-damping \* t) \* std::cos(omega \* t \* timeScale);

return { L \* std::sin(theta\_t),1.8f - L \* std::cos(theta\_t) };

}

Vec3 display(float currentFrame) {

const float L = 2.5f; // Длина маятника (м)

const float g = 9.81f; // Ускорение свободного падения (м/с^2)

const float omega\_e = 2 \* glm::pi<float>() / 286400; //7.2921159e-5; // Угловая скорость вращения Земли (рад/с)

const float latitude = 45.0f; // Широта (градусы)

float omega\_p = sqrt(g / L); // Собственная частота колебаний маятника

float omega\_f = omega\_e \* sin(glm::radians(latitude)); // Частота прецессии Фуко

float A = 0.5f; // Амплитуда

float z0 = L; // Исходная высота маятника

// Основное движение маятника

float x = A \* cos(omega\_p \* currentFrame);

float y = A \* sin(omega\_p \* currentFrame);

float z = z0 - A \* cos(omega\_p \* currentFrame); // Отклонение по оси Z

// Скорость поворота плоскости колебаний (эффект Фуко)

//float a1 = (15.0f / 3600.0f) \* (1.0f - (3.0f / 8.0f) \* pow(sin(glm::radians(latitude)), 2)) \* sin(glm::radians(latitude)) \* glm::pi<float>() / 180.0f;

//float theta = currentFrame \* a1;

// Вращение системы координат

float rotatedX = cos(currentFrame) + 1.5f\*sin(currentFrame);

if (rotatedX < -1.08f) {

rotatedX = -1.08f;

}

float rotatedZ = 0.5f\*cos( currentFrame) + 0.5f \*sin(currentFrame) - 3.0f;

return { rotatedX, z ,rotatedZ };

}

std::vector<float> generateSphereVertices(float radius, int sectorCount, int stackCount) {

std::vector<float> vertices;

float x, y, z, xy; // координаты

float sectorStep = 2 \* 3.14f / sectorCount;

float stackStep = 3.14f / stackCount;

float sectorAngle, stackAngle;

for (int i = 0; i <= stackCount; ++i) {

stackAngle = 3.14f / 2 - i \* stackStep; // От верхнего полюса до нижнего

xy = radius \* cosf(stackAngle); // x, z координаты на сфере

z = radius \* sinf(stackAngle); // y координата на сфере

// Генерация вершин

for (int j = 0; j <= sectorCount; ++j) {

sectorAngle = j \* sectorStep; // Угол сектора

// Вычисляем координаты для каждой вершины

x = xy \* cosf(sectorAngle); // X координата

y = xy \* sinf(sectorAngle); // Z координата

// Нормали для освещения

vertices.push\_back(x);

vertices.push\_back(y);

vertices.push\_back(z);

vertices.push\_back(x / radius); // Нормаль по x

vertices.push\_back(y / radius); // Нормаль по y

vertices.push\_back(z / radius); // Нормаль по z

}

}

return vertices;

}

std::vector<GLuint> generateSphereIndices(int sectorCount, int stackCount) {

std::vector<GLuint> indices;

for (int i = 0; i < stackCount; ++i) {

int k1 = i \* (sectorCount + 1); // Индексы первого ряда

int k2 = (i + 1) \* (sectorCount + 1); // Индексы второго ряда

for (int j = 0; j < sectorCount; ++j) {

if (i != 0) {

indices.push\_back(k1 + j);

indices.push\_back(k2 + j);

indices.push\_back(k1 + j + 1);

}

if (i != (stackCount - 1)) {

indices.push\_back(k1 + j + 1);

indices.push\_back(k2 + j);

indices.push\_back(k2 + j + 1);

}

}

}

return indices;

}

void renderSphere() {

std::vector<float> vertices = generateSphereVertices(0.3f, 36, 18); // Радиус 1, 36 секторов и 18 слоев

std::vector<GLuint> indices = generateSphereIndices(36, 18);

unsigned int VAO, VBO, EBO;

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glGenBuffers(1, &EBO);

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* sizeof(float), &vertices[0], GL\_STATIC\_DRAW);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(GLuint), &indices[0], GL\_STATIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(GLfloat), (GLvoid\*)0);

glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(GLfloat), (GLvoid\*)(3 \* sizeof(GLfloat)));

glEnableVertexAttribArray(1);

glBindVertexArray(0);

glBindVertexArray(VAO);

glDrawElements(GL\_TRIANGLES, indices.size(), GL\_UNSIGNED\_INT, 0);

glBindVertexArray(0);

glDeleteVertexArrays(1, &VAO);

glDeleteBuffers(1, &VBO);

glDeleteBuffers(1, &EBO);

}

void renderPlatform() {

float verticesPlatform[] = {

-0.5f, -0.1f, -0.5f, 0.0f, 0.0f, -1.0f,

0.5f, -0.1f, -0.5f, 0.0f, 0.0f, -1.0f,

0.5f, 0.1f, -0.5f, 0.0f, 0.0f, -1.0f,

0.5f, 0.1f, -0.5f, 0.0f, 0.0f, -1.0f,

-0.5f, 0.1f, -0.5f, 0.0f, 0.0f, -1.0f,

-0.5f, -0.1f, -0.5f, 0.0f, 0.0f, -1.0f,

-0.5f, -0.1f, 0.5f, 0.0f, 0.0f, 1.0f,

0.5f, -0.1f, 0.5f, 0.0f, 0.0f, 1.0f,

0.5f, 0.1f, 0.5f, 0.0f, 0.0f, 1.0f,

0.5f, 0.1f, 0.5f, 0.0f, 0.0f, 1.0f,

-0.5f, 0.1f, 0.5f, 0.0f, 0.0f, 1.0f,

-0.5f, -0.1f, 0.5f, 0.0f, 0.0f, 1.0f,

-0.5f, 0.1f, 0.5f, -1.0f, 0.0f, 0.0f,

-0.5f, 0.1f, -0.5f, -1.0f, 0.0f, 0.0f,

-0.5f, -0.1f, -0.5f, -1.0f, 0.0f, 0.0f,

-0.5f, -0.1f, -0.5f, -1.0f, 0.0f, 0.0f,

-0.5f, -0.1f, 0.5f, -1.0f, 0.0f, 0.0f,

-0.5f, 0.1f, 0.5f, -1.0f, 0.0f, 0.0f,

0.5f, 0.1f, 0.5f, 1.0f, 0.0f, 0.0f,

0.5f, 0.1f, -0.5f, 1.0f, 0.0f, 0.0f,

0.5f, -0.1f, -0.5f, 1.0f, 0.0f, 0.0f,

0.5f, -0.1f, -0.5f, 1.0f, 0.0f, 0.0f,

0.5f, -0.1f, 0.5f, 1.0f, 0.0f, 0.0f,

0.5f, 0.1f, 0.5f, 1.0f, 0.0f, 0.0f,

-0.5f, -0.1f, -0.5f, 0.0f, -1.0f, 0.0f,

0.5f, -0.1f, -0.5f, 0.0f, -1.0f, 0.0f,

0.5f, -0.1f, 0.5f, 0.0f, -1.0f, 0.0f,

0.5f, -0.1f, 0.5f, 0.0f, -1.0f, 0.0f,

-0.5f, -0.1f, 0.5f, 0.0f, -1.0f, 0.0f,

-0.5f, -0.1f, -0.5f, 0.0f, -1.0f, 0.0f,

-0.5f, 0.1f, -0.5f, 0.0f, 1.0f, 0.0f,

0.5f, 0.1f, -0.5f, 0.0f, 1.0f, 0.0f,

0.5f, 0.1f, 0.5f, 0.0f, 1.0f, 0.0f,

0.5f, 0.1f, 0.5f, 0.0f, 1.0f, 0.0f,

-0.5f, 0.1f, 0.5f, 0.0f, 1.0f, 0.0f,

-0.5f, 0.1f, -0.5f, 0.0f, 1.0f, 0.0f

};

unsigned int platformVAO, platformVBO;

glGenVertexArrays(1, &platformVAO);

glBindVertexArray(platformVAO);

glGenBuffers(1, &platformVBO);

glBindBuffer(GL\_ARRAY\_BUFFER, platformVBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(verticesPlatform), verticesPlatform, GL\_STATIC\_DRAW);

// Координатные атрибуты

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)0);

glEnableVertexAttribArray(0);

// Атрибуты нормалей

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(float), (void\*)(3 \* sizeof(float)));

glEnableVertexAttribArray(1);

glBindVertexArray(platformVAO);

glDrawArrays(GL\_TRIANGLES, 0, 36);

glBindVertexArray(0);

glDeleteVertexArrays(1, &platformVAO);

glDeleteBuffers(1, &platformVBO);

}

void renderLine(glm::vec3 pendulumPos) {

unsigned int lineVAO, lineVBO;

glGenVertexArrays(1, &lineVAO);

glGenBuffers(1, &lineVBO);

glBindVertexArray(lineVAO);

glBindBuffer(GL\_ARRAY\_BUFFER, lineVBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(glm::vec3) \* 2, NULL, GL\_DYNAMIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, (void\*)0);

glEnableVertexAttribArray(0);

glm::vec3 lineVertices[] = { glm::vec3(0.0f, 1.39f, -1.0f), pendulumPos };

glBindBuffer(GL\_ARRAY\_BUFFER, lineVBO);

glBufferSubData(GL\_ARRAY\_BUFFER, 0, sizeof(lineVertices), lineVertices);

glBindVertexArray(lineVAO);

glDrawArrays(GL\_LINES, 0, 2);

}

void renderPoint(glm::vec3 pendulumPos) {

glm::vec3 linePoints[] = { pendulumPos };

unsigned int pointVAO, pointVBO;

glGenVertexArrays(1, &pointVAO);

glGenBuffers(1, &pointVBO);

glBindVertexArray(pointVAO);

glBindBuffer(GL\_ARRAY\_BUFFER, pointVBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(linePoints), linePoints, GL\_DYNAMIC\_DRAW);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);

glEnableVertexAttribArray(0);

glPointSize(3.0f);

glBindVertexArray(pointVAO);

glDrawArrays(GL\_POINTS, 0, 1);

glDeleteVertexArrays(1, &pointVAO);

glDeleteBuffers(1, &pointVBO);

}

std::array<Shader, 4> createShaders() {

return { Shader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/shadow\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/shadow\_fs.txt"),

Shader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/quad\_depth\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/quad\_depth\_fs.txt"),

Shader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/mapping\_depth\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/mapping\_depth\_fs.txt"),

Shader("C:/Users/pm14g/dev/Glitter/Glitter/Shaders/point\_vs.txt", "C:/Users/pm14g/dev/Glitter/Glitter/Shaders/point\_fs.txt")

};

}

// renderCube() рендерит 1x1 3D-ящик в NDC

unsigned int cubeVAO = 0;

unsigned int cubeVBO = 0;

void renderCube()

{

// Инициализация (если необходимо)

if (cubeVAO == 0)

{

float vertices[] = {

// задняя грань

-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f, // нижняя-левая

1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f, // верхняя-правая

1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f, // нижняя-правая

1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f, // верхняя-правая

-1.0f, -1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f, // нижняя-левая

-1.0f, 1.0f, -1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f, // верхняя-левая

// передняя грань

-1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // нижняя-левая

1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, // нижняя-правая

1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // верхняя-правая

1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // верхняя-правая

-1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, // верхняя-левая

-1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // нижняя-левая

// грань слева

-1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f, // верхняя-правая

-1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // верхняя-левая

-1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, // нижняя-левая

-1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, // нижняя-левая

-1.0f, -1.0f, 1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // нижняя-правая

-1.0f, 1.0f, 1.0f, -1.0f, 0.0f, 0.0f, 1.0f, 0.0f, // верхняя-правая

// грань справа

1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, // верхняя-левая

1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, // нижняя-правая

1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // верхняя-правая

1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, // нижняя-правая

1.0f, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, // верхняя-левая

1.0f, -1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // нижняя-левая

// нижняя грань

-1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f, 0.0f, 1.0f, // верхняя-правая

1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f, 1.0f, 1.0f, // верхняя-левая

1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f, // нижняя-левая

1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f, // нижняя-левая

-1.0f, -1.0f, 1.0f, 0.0f, -1.0f, 0.0f, 0.0f, 0.0f, // нижняя-правая

-1.0f, -1.0f, -1.0f, 0.0f, -1.0f, 0.0f, 0.0f, 1.0f, // верхняя-правая

// верхняя грань

-1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, // верхняя-левая

1.0f, 1.0f , 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // нижняя-правая

1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, // верхняя-правая

1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // нижняя-правая

-1.0f, 1.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, // верхняя-левая

-1.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f // нижняя-левая

};

glGenVertexArrays(1, &cubeVAO);

glGenBuffers(1, &cubeVBO);

// Заполняем буфер

glBindBuffer(GL\_ARRAY\_BUFFER, cubeVBO);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);

// Связываем вершинные атрибуты

glBindVertexArray(cubeVAO);

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)0);

glEnableVertexAttribArray(1);

glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(3 \* sizeof(float)));

glEnableVertexAttribArray(2);

glVertexAttribPointer(2, 2, GL\_FLOAT, GL\_FALSE, 8 \* sizeof(float), (void\*)(6 \* sizeof(float)));

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

}

// Рендерим ящик

glBindVertexArray(cubeVAO);

glDrawArrays(GL\_TRIANGLES, 0, 36);

glBindVertexArray(0);

}

void processInput(GLFWwindow\* window, Shader shader, Shader debugDepthQuad,

Shader simpleDepthShader, Shader shaderPoint)

{

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)

camera.ProcessKeyboard(FORWARD, deltaTime);

if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)

camera.ProcessKeyboard(BACKWARD, deltaTime);

if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)

camera.ProcessKeyboard(LEFT, deltaTime);

if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)

camera.ProcessKeyboard(RIGHT, deltaTime);

if (glfwGetKey(window, GLFW\_KEY\_Q) == GLFW\_PRESS)

camera.Position = glm::vec3(0.0f, 0.0f, 5.0f);

if (glfwGetKey(window, GLFW\_KEY\_P) == GLFW\_PRESS)

point = false;

if (glfwGetKey(window, GLFW\_KEY\_O) == GLFW\_PRESS)

point = true;

if (glfwGetKey(window, GLFW\_KEY\_I) == GLFW\_PRESS)

ckle(window, shader, debugDepthQuad, simpleDepthShader, shaderPoint);

}

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height)

{

SCR\_WIDTH = width;

SCR\_HEIGHT = height;

glViewport(0, 0, width, height);

}

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos)

{

if (firstMouse)

{

lastX = xpos;

lastY = ypos;

firstMouse = false;

}

float xoffset = xpos - lastX;

float yoffset = lastY - ypos; // перевернуто, так как y-координаты идут снизу вверх

lastX = xpos;

lastY = ypos;

camera.ProcessMouseMovement(xoffset, yoffset);

}

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)

{

camera.ProcessMouseScroll(yoffset);

}

// Вспомогательная функция загрузки 2D-текстур из файла

unsigned int loadTexture(char const\* path)

{

unsigned int textureID;

glGenTextures(1, &textureID);

int width, height, nrComponents;

unsigned char\* data = stbi\_load(path, &width, &height, &nrComponents, 0);

if (data)

{

GLenum format;

if (nrComponents == 1)

format = GL\_RED;

else if (nrComponents == 3)

format = GL\_RGB;

else if (nrComponents == 4)

format = GL\_RGBA;

glBindTexture(GL\_TEXTURE\_2D, textureID);

glTexImage2D(GL\_TEXTURE\_2D, 0, format, width, height, 0, format, GL\_UNSIGNED\_BYTE, data);

glGenerateMipmap(GL\_TEXTURE\_2D);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, format == GL\_RGBA ? GL\_CLAMP\_TO\_EDGE : GL\_REPEAT); // для данного урока: испольщуйте GL\_CLAMP\_TO\_EDGE для предотвращения возникновения полупрозрачных границ. Благодаря интерполяции берутся тексели из следующего повторения

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, format == GL\_RGBA ? GL\_CLAMP\_TO\_EDGE : GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

stbi\_image\_free(data);

}

else

{

std::cout << "Texture failed to load at path: " << path << std::endl;

stbi\_image\_free(data);

}

return textureID;

}