**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АМ**

Курсовая РАБОТА

**по дисциплине «Дифференциальные уравнения»**

Тема: Симуляция поведения айсберга, падающего в воду

|  |  |  |
| --- | --- | --- |
| Студентка гр. 8382 |  | Кулачкова М.К. |
| Студент гр. 8382 |  | Мирончик П.Д. |
| Преподаватель |  | Павлов Д.А. |

Санкт-Петербург

2021

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студенты: Кулачкова М.К., Мирончик П.Д. | | |
| Группа 8382 | | |
| Тема работы: Симуляция поведения айсберга, падающего в воду | | |
| Исходные данные:  Набором точек задается выпуклое трехмерное тело с плотностью меньшей, чем плотность воды. Необходимо смоделировать поведение этого тела при падении в воду. | | |
| Содержание пояснительной записки:  «Содержание», «Введение», «Представление тела», «Решение задачи Коши», «Визуализация», «Заключение», «Список использованных источников» | | |
| Предполагаемый объем пояснительной записки:  Не менее 20 страниц. | | |
| Дата выдачи задания: 14.04.2021 | | |
| Дата сдачи реферата: 29.05.2021 | | |
| Дата защиты реферата: 00.00.2000 | | |
| Студентка |  | Кулачкова М.К. |
| Студент |  | Мирончик П.Д. |
| Преподаватель |  | Павлов Д.А. |

**Аннотация**

В данной работе осуществляется симуляция поведения айсберга, падающего в воду. Айсберг представляет собой выпуклое тело, задаваемое набором вершин и имеющее плотность меньшую, чем плотность воды. Поведение айсберга описывается системой ОДУ, которая численно решается классическим методом Рунге-Кутты 4-го порядка. Моделируемая динамическая система визуализируется с использованием библиотеки OpenGL и фреймворка Qt.

**Summary**

In this project we simulate the behavior of an iceberg falling into the water. The iceberg is a convex body defined by a set of vertices and its density is lower than the density of water. The behavior of the iceberg is described using a system of ordinary differential equations which is solved numerically using classic fourth order Runghe-Kutta method. The simulated dynamic system is visualized using the OpenGL library and the Qt framework.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Представление тела | 6 |
| 1.1. | Задание формы | 6 |
| 1.2 | Загрузка фигуры | 6 |
| 1.3. | Состояние | 10 |
| 2. | Решение задачи Коши | 12 |
| 2.1. | Производная состояния | 12 |
| 2.2. | Определение части тела, погруженной в воду | 13 |
| 2.3. | Вычисление нового состояния | 16 |
| 2.4. | Изменение состояния | 17 |
| 3. | Визуализация | 18 |
| 3.1. | Drawer | 18 |
| 3.2. | Figure | 19 |
| 3.3. | ViewMatrixWrapper | 19 |
| 3.4. | FigureWrapper | 20 |
| 3.5. | VerticesStretcher | 21 |
|  | Заключение | 22 |
|  | Список использованных источников | 23 |
|  | Приложение А. Иллюстрация работы приложения | 24 |

**введение**

Целью работы является симуляция и визуализация поведения айсберга (трехмерного тела с плотностью ниже плотности воды) при падении в воду. Так как поведение айсберга задается системой обыкновенных дифференциальных уравнений, задача сводится к численному решению задачи Коши. В данной работе для решения задачи Коши используется классический метод Рунге-Кутты 4-го порядка.

Визуализация динамической системы заключается в построении анимированной сцены, изображающей движение айсберга. Программа также снабжается пользовательским интерфейсом: можно изменить точку или угол обзора сцены, загрузить фигуру для симуляции из файла, изменить начальную высоту фигуры, сделать один шаг симуляции с заданным промежутком.

**1. Представление тела**

**1.1. Задание формы**

Айсберг задается набором граней, каждая из которых, в свою очередь, задается набором вершин. Реализован метод, вычисляющий нормаль к грани, что необходимо при вычислении различных характеристик тела, а также для его корректной отрисовки. Чтобы вычисляемая нормаль была внешней, вносится условие на порядок следования вершин в грани: вершины должны задаваться против часовой стрелки.

По умолчанию айсберг имеет кубическую форму, но можно задать другую форму, загрузив набор вершин из текстового файла. Файл, задающий новую фигуру, должен начинаться с количества вершин многогранника, а затем содержать список координат вершин.

**1.2. Загрузка фигуры**

Для создания объекта требуется знать лишь набор вершин, к которым выдвигается единственное условие: они не должны лежать в одной плоскости. Алгоритм или его отдельные части, вероятнее всего, имеют какое-то название, однако при реализации в данной работе он придумывался с нуля, поэтому будем называть его “натягиванием поверхности”. Далее приведено описание алгоритма (конкретную реализацию можно найти в файлах *verticesstretcher.h* и *verticesstretcher.cpp*).

Рассмотрим шаг . Имеется набор натянутых плоскостей. Каждая плоскость имеет направление обхода против часовой стрелки и образует набор направленных ребер. Вместе все плоскости дают набор ребер, в котором некоторые ребра имеют, а некоторые не имеют противоположно направленного “соседа”.

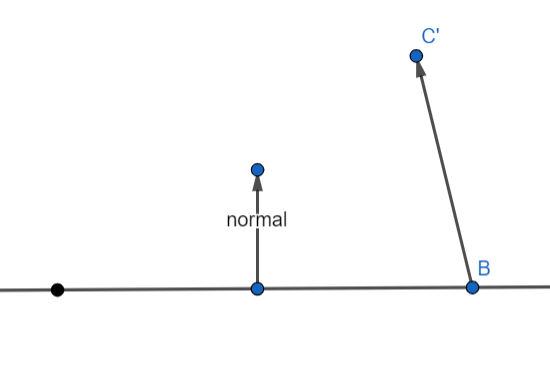
*Под натянутыми плоскостями подразумеваются грани выпуклой фигуры.*

*Для ребра противоположно направленным будет являться , где - точки, полученные из входного табора вершин.*

Из этого набора ребер берется одно ребро, не имеющее “соседа”, и противоположное к нему ребро принимается как одна сторона генерируемой плоскости, которая так же будет являться частью натянутой поверхности. Обозначим это ребро как *AB*.

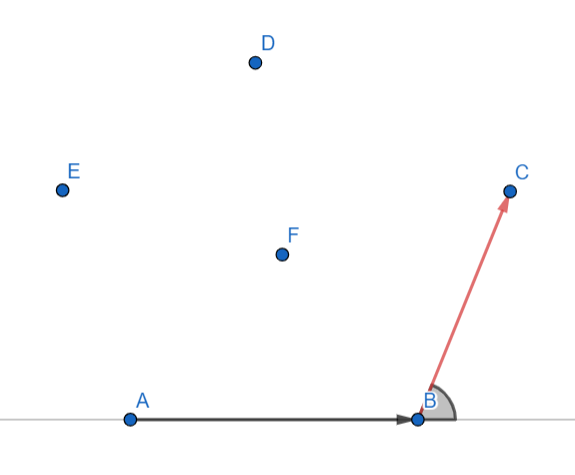
Теперь представим ребро в качестве оси, на которой поворачивается плоскость. Наша задача подобрать третью точку так, чтобы с лицевой стороны плоскости не оказалось ни одной точки - в этом случае она может считаться частью поверхности. Для этого мы проходим по всем входным вершинам, и, если обнаруживаем, что текущая вершина находится с лицевой стороны поверхности, “поворачиваем” поверхность вокруг оси *AB* так, чтобы она содержала вершину.

*Определить, находится ли точка с лицевой стороны, достаточно просто. Т.к. наша плоскость характеризуется тремя точками: A, B и C (где C - это вершина, к которой “повернута” плоскость), и известно, что лицевой является сторона, где порядок обхода вершин A,B,C направлен против часовой стрелки, можно однозначно сформировать вектор нормали к плоскости, который будет иметь направление от плоскости в лицевое полупространство. Далее нам достаточно получить скалярное произведение вектора нормали на вектор , где B - любая точка плоскости, - проверяемая точка. Если скалярное произведение больше нуля - точка лежит в лицевой полуплоскости и плоскость нужно довернуть.*



Теперь мы имеем плоскость, которая содержит как минимум 3 вершины, причем при объединении всех вершин этой плоскости получится одна из поверхностей выпуклого многоугольника. Осталось только сформировать из этих точек выпуклую двумерную фигуру (двумерную, т.к. все точки лежат в одной плоскости).

Рассмотрим следующую ситуацию:



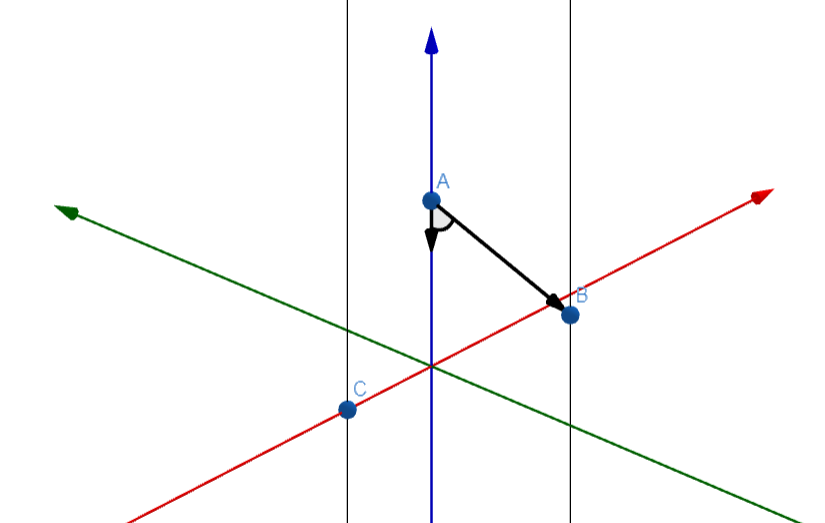
Известно, что все точки лежат в одной плоскости, а вектор AB гарантированно входит во внешний контур. Задача поиска следующего вектора контура () решается все тем же скалярным произведением. Фактически нам требуется найти косинусы углов между векторами и , и вектор, косинус которого с будет наибольшим, и будет следующим вектором контура. На каждом следующем шаге вместо вектора выбирается последний добавленный вектор контура до тех пор, пока контур не замкнется. После замыкания контура полученную плоскость можно добавлять в набор “натянутых” плоскостей.

Алгоритм завершится тогда, когда не останется ни одного ребра, не имеющего противоположно направленного соседа.

Итак, осталась лишь самая малость: как получить ось плоскости на первом шаге, когда список плоскостей еще пуст и нет ребер? Ответ весьма прост: так же, как ищется следующее ребро в формировании контура поверхности, но в трехмерном формате.

Во-первых, выберем точку, имеющую наибольшее значение по оси *z* и обозначим ее как *A*. Очевидно, что она будет входить в поверхность трехмерной фигуры.

Далее найдем такую вершину *B*, чтобы вектор *AB* имел наибольший угол с вектором . Полученный вектор *AB* и будет являться искомым первым ребром для построения поверхности. При этом направление ребра не имеет значения, т.к. в конечном итоге все равно будет использован его противоположно направленный “сосед”.



Таким образом, для сохранения и восстановления фигуры нет необходимости в сохранении набора ребер или плоскостей, достаточно лишь записать все уникальные вершины этой фигуры. Более того, появляется возможность случайной генерации фигур - парсер, написанный по такому алгоритму, способен работать с любым, даже содержащим “лишние”, расположенными внутри фигуры, вершинами. Единственное условие, которое должно быть соблюдено - вершины не должны лежать в одной плоскости.

**1.3. Состояние**

Характеристики движения и положение тела в пространстве в каждый момент времени можно определить следующим тензором состояния:

где – центр масс тела, который определяет смещение тела относительно начала координат, – его импульс, который определяет линейную скорость движения тела, – матрица поворота, которая определяет поворот тела относительно своего изначального положения, – момент импульса тела, определяющий угловую скорость вращения тела.

Для сокращения погрешности вычислений вместо матрицы поворота было решено использовать кватернион вращения, который представляет собой вектор из четырех элементов

где – угол поворота тела, – ось вращения. Матрица поворота выражается через кватернион вращения следующим образом:

Таким образом, состояние тела задается тензором

В начальный момент времени векторы и равны нулевому вектору, , а вектор вычисляется по алгоритму, описанному Б. Миртичем.

Состояние хранится в виде полей класса, который задает айсберг, но для упрощения работы с ним была создана отдельная структура, которая передается в функции, каким-либо образом изменяющие состояние.

**2. Решение задачи Коши**

**2. 1. Производная состояния**

Для построения задачи Коши необходимо определить производную состояния – функцию . Если состояние тела в момент времени равно

то производная состояния по времени равна

Масса тела вычисляется по алгоритму Б. Миртича.

– равнодействующая сил, действующих на тело. В данной задаче равна , где – плотность морской воды, – объем части тела, погруженной в воду, – ускорение свободного падения. Процесс вычисления объема тела, погруженного в воде, описан в пункте 1.4. В реальности движению айсберга в воде препятствует сила сопротивления жидкости. Вычисление ее в данной работе было бы слишком сложным и выходит за рамки решаемой задачи, поэтому было введено «фиктивное» сопротивление, равное 0.2, которое отнимается от величины при вычислении производной состояния.

Производная кватерниона вращения равна , где – кватернион, получаемый из вектора угловой скорости вращения тела, которая, в свою, очередь, вычисляется как , – тензор инерции тела. Тензор инерции при изначальном определении тела вычисляется по алгоритму Б. Миртича, а при последующих изменениях состояния тела по формуле . В формуле производной присутствует произведение кватернионов. Кватернионы и можно определить следующим образом: , . Тогда их произведение равно .

– момент сил, действующих на тело. Вычисляется по формуле . Так как в данной динамической системе на тело действуют всего две силы, причем одна из них приложена к его центру масс, момент сил равен , где – центр масс части тела, погруженной в воду. Было также введено фиктивное сопротивление воды вращению тела, равное , которое вычитается из полученного вектора момента сил.

Ниже приведена реализация метода, вычисляющего производную.

void Polyhedron::x\_dot(State state, State& state\_dot)

{

Polyhedron \* tmp = *new* Polyhedron(*this*->faces);

tmp->setState(state);

state\_dot.c = tmp->p/tmp->mass;

state\_dot.p = tmp->calculateForces();

QVector3D tmpOmega = tmp->calculateOmega();

Quaternion qOmega(0, tmpOmega);

state\_dot.q = (qOmega \* tmp->q) / 2.0f;

state\_dot.L = tmp->calculateTau();

*delete* tmp;

}

Так как производная должна быть функцией от состояния, однако зависит от формы тела, которая в состояние не входит, внутри метода приходится создавать временный экземпляр айсберга с такой же формой, как у текущего, у которого устанавливается передаваемое в функцию состояние. Для этого временного экземпляра затем вычисляются входящие в производную переменные.

**2.2. Определение части тела, погруженной в воду**

Для вычисления производной состояния тела необходимо определить характеристики части тела, погруженной в воду. Погруженная в воду часть тела представляется в виде отдельного многогранника, для чего требуется определить грани, из которых этот многогранник будет состоять.

В цикле обходятся все вершины исходного многогранника. Так как поверхность воды считается параллельной координатной плоскости OXY и задается координатой , для того, чтобы определить, находится вершина многогранника под или над водой, достаточно сравнить z-координату вершины с переменной , задающей уровень воды. Вершины, оказавшиеся под водой (z-координата которых меньше или равна значению переменной *z*) добавляются в новую грань.

Если текущая рассматриваемая вершина находится над водой, а следующая за ней – под водой, или наоборот, соединяющее их ребро пересекается прямой, соответствующей уровню воду, и необходимо найти точку пересечения. Обозначим текущую вершину , а следующую за ней – . Уравнение прямой, проходящей через эти две точки, имеет вид: . Коэффициент вычисляется по формуле . Тогда координаты точки пересечения равны . Эта точка также добавляется в новую грань. После обхода всех вершин из точек, находящихся под водой и на поверхности создается грань, которая попадает в массив граней, из которых затем будет сформирован многогранник.

Далее необходимо сформировать ту грань многогранника, которая получается пересечением исходного многогранника поверхностью воды. Алгоритм нахождения точек пересечения был описан выше, однако для корректной работы программы необходимо, чтобы вершины в грани обходились строго против часовой стрелки. Рассмотрим случай, когда поверхность воды пересекает грань исходного многогранника в точках, не совпадающих с ее вершинами, поскольку другие случаи можно свести к этому. В процессе анализа задачи было замечено, что, если при обходе вершин в грани рассматриваемая вершина находится над водой, а следующая за ней – под водой, то точка пересечения с поверхностью воды, лежащая на соединяющем их ребре, должна при задании грани, лежащей в плоскости поверхности воды, идти раньше, чем точка пересечения, которая получится при движении из-под воды наружу (см. рис. 1). Пара этих точек пересечения записывается в массив в том порядке, в котором они должны попасть в искомую грань, этот массив добавляется в общий массив ребер, лежащих на поверхности воды.

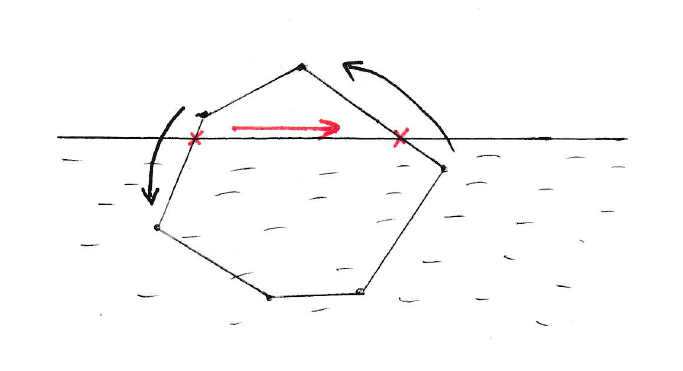


Рисунок 1 – Схема обхода вершин

После обхода всех граней исходного многогранника из массива ребер необходимо собрать грань. Если грань пересекается водой в одной точке, то начало и конец ребра, полученного при обходе этой грани, совпадают. Но эта точка также должна входить как минимум в два других ребра, поэтому на первом шаге построения грани из ребер она будет игнорироваться, а далее такие ребра с одинаковыми концами будут удаляться из массива. На первом шаге построения грани находится первое ребро в массиве, концы которого не равны. Вершины на его концах добавляются в массив вершин, из которых впоследствии будет сформирована грань, а само ребро удаляется из массива. Бывают ситуации, когда таких ребер не существует. В таком случае грань не может быть построена. Следующая последовательность действий повторяется до тех пор, пока первая и последняя вершины в массиве вершин не совпадут или пока из массива ребер не будут удалены все элементы:

1. Обходятся все ребра в массиве;
2. Если начало и конец ребра совпадают, ребро удаляется из массива;
3. Если начало текущего ребра совпадает с последним элементом в массиве вершин, его конец добавляется в массив вершин и ребро удаляется из массива ребер.

Если после окончания работы алгоритма первая и последняя вершины в массиве совпадают, последняя вершина удаляется во избежание дублирования и на основе полученного массива строится грань. Иначе грань оказывается незамкнутой, все вершины удаляются из списка и возвращается пустая грань.

Если полученная грань непустая, она добавляется в массив граней, из которого создается многогранник. При инициализации у многогранника по алгоритму Б. Миртича вычисляются центр масс и масса. Объем многогранника вычисляется как масса, деленная на плотность. Также по формуле вычисляется действующая на тело сила Архимеда. Вычисленные величины записываются в соответствующие поля исходного многогранника для дальнейшего использования при вычислении производной состояния.

**2.3. Вычисление нового состояния**

Для вычисления нового состояния тела используется классический метод Рунге-Кутты 4-го порядка:

Так как вычисление производной состояния привязано к форме айсберга, полностью отделить метод численного интегрирования от реализации айсберга не представляется возможным. Реализация метода, осуществляющего вычисление нового состояния, приведена ниже.

void Polyhedron::computeNewState(float h) {

State oldState(*this*->c, *this*->p, *this*->q, *this*->L);

State k1, k2, k3, k4;

x\_dot(oldState, *k1*);

x\_dot(oldState + h/2.0f\*k1, *k2*);

x\_dot(oldState + h/2.0f\*k2, *k3*);

x\_dot(oldState + h\*k3, *k4*);

State newState = oldState + h\*(k1/6.0f + k2/3.0f + k3/3.0f + k4/6.0f);

setState(newState);

}

**2.4. Изменение состояния**

После вычисления нового состояния необходимо изменить координаты вершин айсберга в соответствии с этим состоянием, а также сохранить это состояние в полях объекта, определяющего айсберг.

На координаты вершин влияют только центр масс и матрица поворота, которая вычисляется из кватерниона вращения. Сначала необходимо вычислить координаты вершин в локальной системе координат. Для этого из координат вершин вычитаются координаты текущего центра масс, тем самым переводя их из глобальной системы отсчета в промежуточную, а затем полученные координаты умножаются слева на матрицу, обратную текущей матрице поворота. Координаты в локальной системе координат умножаются на новую матрицу поворота, а затем к ним прибавляется новый вектор центра масс. Получили координаты вершин айсберга в глобальной системе координат, которые используются при вычислении производной, а также при визуализации динамической системы.

**3. Визуализация**

Для визуализации использовался интерфейс OpenGL. Он предоставляет набор низкоуровневых операций, позволяющих оперировать простейшими фигурами, работать с освещением, цветами фигур и т.п. Во многом визуализация айсберга сводится именно к особенностям использования данного интерфейса, поэтому в 3 разделе представлено краткое описание классов, позволяющее понять связи между объектами и задачи каждого из объектов.

**3.1. Drawer**

class Drawer : public QOpenGLWidget

Корневой узел объектов, при помощи которых производится отображение сцены. Задачи Drawer:

1. Настройка openGL. Drawer задает основные параметры, такие как:

* glEnable: GL\_ALPHA\_TEST, GL\_DEPTH\_TEST, GL\_LIGHTING, GL\_NORMALIZE, GL\_BLEND;
* Установка glBlendFunc;
* Очищение буфера цвета и глубины;
* Установка критерия лицевой поверхности граней glFrontFace;
* Включение режима двустороннего освещения glLightModelf(GL\_LIGHT\_MODEL\_TWO\_SIDE, GL\_TRUE;
* Установка матриц модели (индивидуально для каждой фигуры), вида и проекции.

2. Освещение. Drawer содержит массив объектов LightConfig и применяет их перед началом отображение фигур Figure.

3. Фигуры. Drawer содержит массив объектов Figure и отвечает за их корректную инициализацию, отображение и деинициализацию, гарантируя корректность порядка вызовов соответствующих методов и валидность контекста во время вызовов.

4. Клавиатура и мышь. Drawer не имеет средств для обработки событий, однако он обеспечивает передачу событий в привязанные к нему ViewMatrixWrapper и FigureWrapper, отслеживая изменение их состояний и выполняя обновление экрана при необходимости.

**3.2. Figure**

class Figure : public QObject

Класс, отвечающий за передачу в openGL данных о гранях и параметров их материала. Задачи Figure:

1. Буферизация. Figure использует стандартный механизм буферов GL\_ARRAY\_BUFFER, GL\_ELEMENT\_ARRAY\_BUFFER, GL\_VERTEX\_ARRAY, GL\_NORMAL\_ARRAY. Основываясь на функциях жизненного цикла attach, paint и detach, вызываемых связанного объекта Drawer, Figure создает, заполняет, использует и удаляет буферы их видеопамяти. При этом Figure поддерживает механизм обновления буферов при изменении списка вершин с использованием соответствующего метода markVertexChanged.

2. Матрица модели. Figure хранит данные о матрице модели фигуры, грани которой отображает. Эта матрица может меняться, после чего необходимо вызвать Figure::markNeedsPaint для уведомления связанного объекта Drawer о наличии изменений и необходимости обновления сцены.

3. Разбиение граней для более реалистичного освещения. Figure предоставляет возможность разбить имеющиеся грани на более мелкие с целью увеличения реалистичности освещения: стандартные шейдеры openGL работают с освещением на уровне вершинного шейдера, поэтому большее количество вершин даст более детализированную картину освещения.

**3.3. ViewMatrixWrapper**

class ViewMatrixWrapper : public QObject

Получает данные о событиях мыши и клавиатуры и изменяет видовую матрицу в соответствии с полученными данными. Алгоритм работы этого класса достаточно прост. Для хранения текущего состояния используются 3 вектора:

\_playerAt - позиция наблюдателя;

\_lookVector - направление взгляда наблюдателя;

\_upVector - вектор, определяющий направление ‘вверх’.

Заметно, что данные вектора практически полностью дублируют параметры, передаваемые в функцию lookAt. Из этого очевидно, что для создания видовой матрицы достаточно вызвать:

lookAt(\_playerAt, \_playerAt + \_lookVector, \_upVector);

При этом изменение позиции и направление взгляда наблюдателя также представляют собой набор простых операций:

1. Поворот вправо-влево равносилен повороту относительно оси \_upVector;

2. Поворот вверх-вниз равносилен повороту относительно оси cross(\_lookVector, \_upVector);

3. Движение в разные стороны представляет смещение точки \_playerAt в направлении вектора, вычисленного при помощи \_lookVector и \_upVector.

Как было замечено выше, события мыши и клавиатуры ViewMatrixWrapper получает непосредственно из связанного объекта Drawer.

**3.4. FigureWrapper**

class FigureWrapper : public QObject

Класс, позволяющий поворачивать фигуру с использованием мыши. События мыши получаются из Drawer-а, аналогично классу ViewMatrixWrapper. К объекту FigureWrapper привязывается объект ViewMatrixWrapper, т.к. для корректного поворота необходимо знать положение камеры.

Алгоритм поворота достаточно простой. Перемещение мыши разбивается на составляющие x и y. Вспомним, что поворот выполняется относительно какой-либо оси, и из этого попробуем повернуть фигуру отдельно по каждому из направлений.

Поворот по x представляет собой поворот относительно вектора вверх, заданного в объекте ViewMatrixWrapper.

Поворот по y можно представить как поворот относительно горизонтального вектора, параллельного экрану. Таким вектором является результат векторного произведения вектора взгляда на вектор вверх из того же ViewMatrixWrapper.

Разумеется, перед началом поворота начало координат находится в центре фигуры - геометрически это середина между минимальными и максимальными значениями вершин объекта.

**3.5. VerticesStretcher**

class VerticesStretcher

Вспомогательный класс, обеспечивающий натягивание поверхности на набор вершин. В нем можно найти реализацию алгоритма натягивания поверхности, подробно описанного в разделе 1.2.

**заключение**

В курсовой работе была осуществлена симуляция поведения айсберга, падающего в воду. Айсберг представляет собой выпуклый многогранник и задается своими вершинами. Состояние айсберга в каждый момент времени определяется тензором

Производная состояния равна

Численное решение полученной системы ОДУ осуществляется при помощи метода Рунге-Кутты 4-го порядка.

Моделируемый процесс визуализируется с использованием графической библиотеки OpenGL. Визуализация снабжена пользовательским интерфейсом: пользователь может изменить точку и угол обзора сцены, сохранить текущую фигуру, загрузить новую, запустить или остановить симуляцию, сделать один шаг метода. Реализация интерфейса осуществлялась при помощи фреймворка Qt.

**список использованных источников**

1. David Baraff. Physically Based Modeling. Rigid Body Simulation: SIGGRAPH 2001 Course Notes
2. Статья о кватернионах // Библиотека CH Robotics. URL: <http://www.chrobotics.com/library/understanding-quaternions> (дата обращения: 17.05.2021)
3. Brian Mirtich. Fast and Accurate Computation of Polyhedral Mass Properties. University of California at Berkeley. 1996.
4. Код алгоритма Б. Миртича для вычисления характеристик многогранника // GitHub. URL: <https://github.com/OpenFOAM/OpenFOAM-4.x/tree/master/src/meshTools/momentOfInertia/volumeIntegration> (дата обращения: 25.04.2021)
5. Список методов Рунге-Кутты // Википедия. URL: <https://en.wikipedia.org/wiki/List_of_Runge%E2%80%93Kutta_methods> (дата обращения: 10.05.2021)

**приложение А**

**Иллюстрация работы приложения**

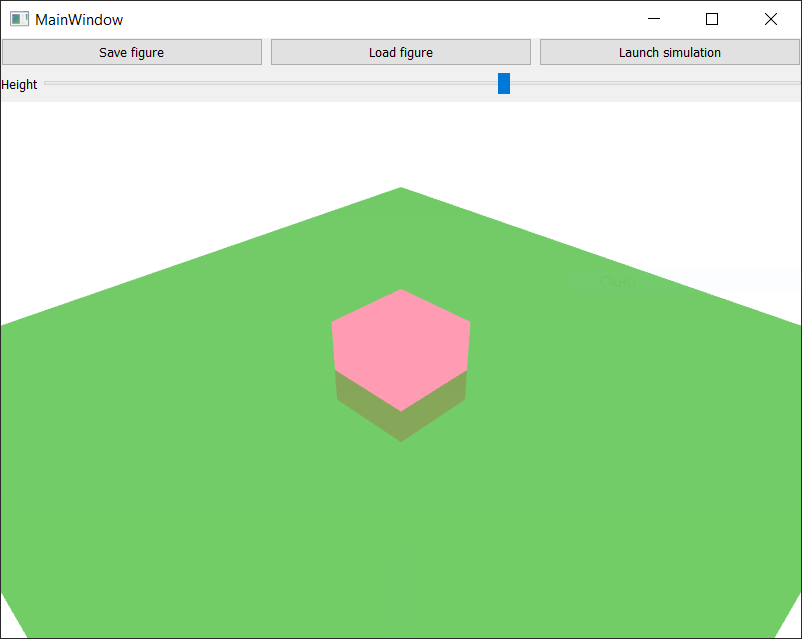


Рисунок 2 – Стартовое окно приложения

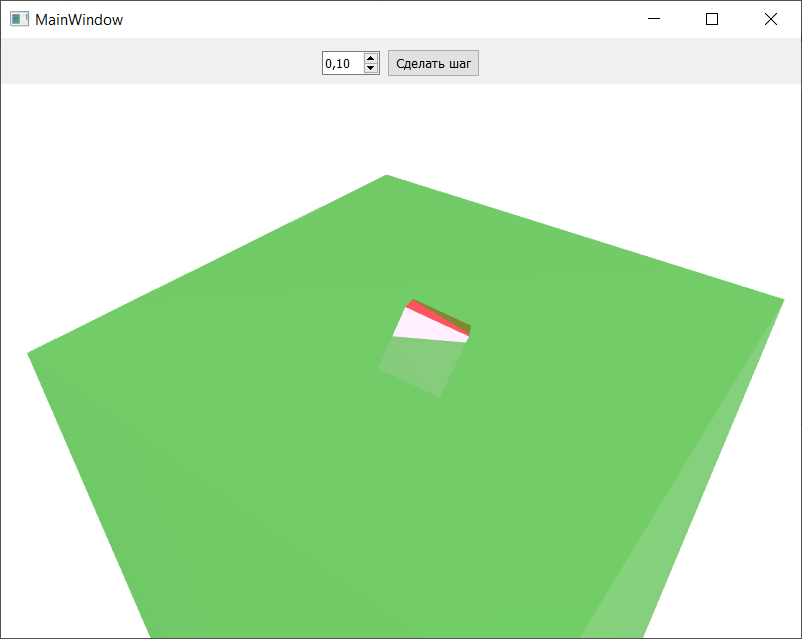


Рисунок 3 – Окно симуляции