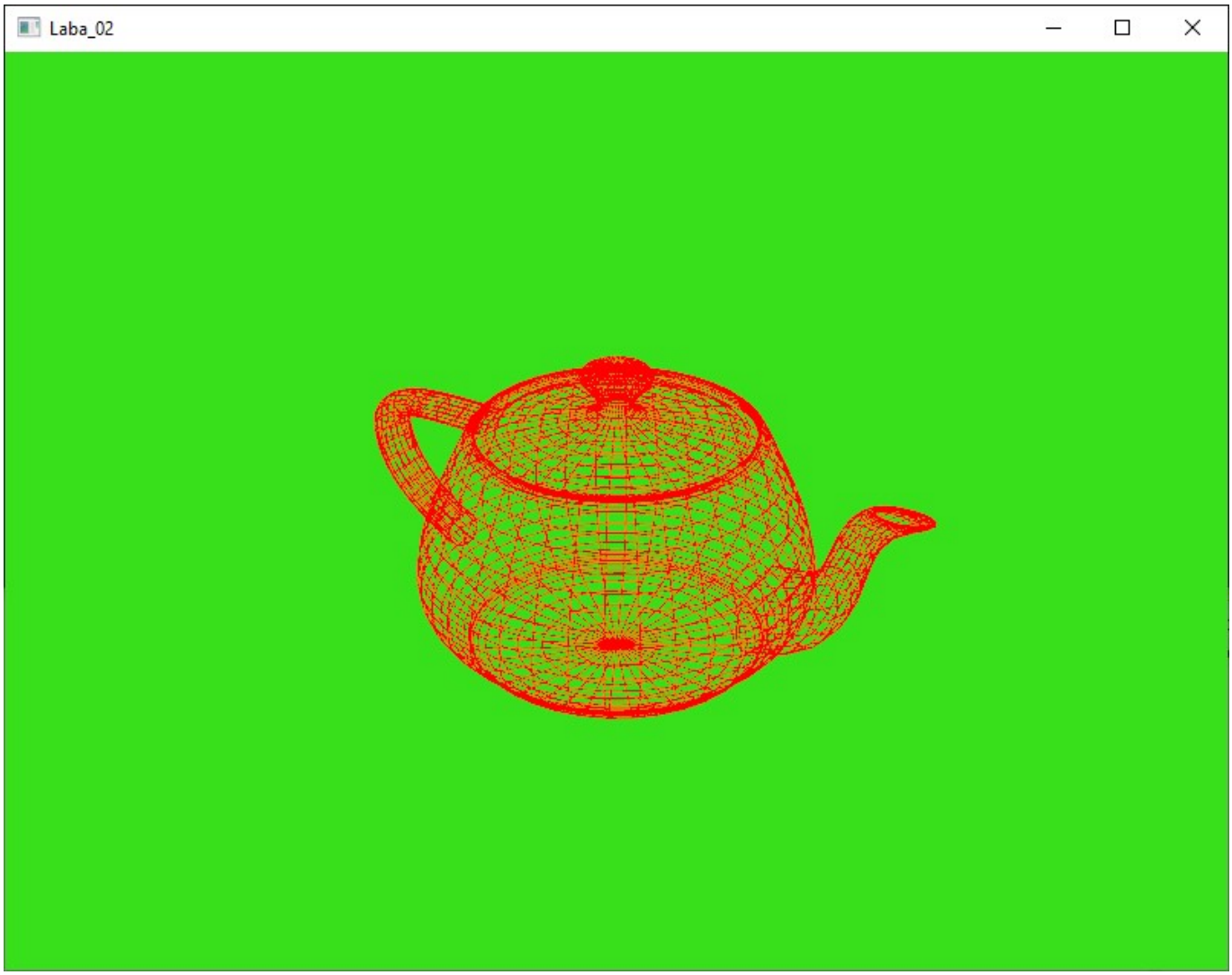
Лабораторная работа №2

Использование библиотеки GLM.

Вторая лабораторная работа призвана познакомить студентов с библиотекой GLM, которая широко используется совместно с OpenGL при программировании трехмерной графики. Библиотека предлагает широкий спектр типов данных для представления векторов и матриц, а также богатый набор функций для осуществления основных операций над ними.

Кроме того, в рамках лабораторной работы необходимо организовать автоматическую смену цветов объекта. Цвет объекта меняется в соответствии с заданным списком цветов через каждую секунду без каких-либо дополнительных действий пользователя. По достижению конца списка, заново устанавливается начальный цвет и смена цветов продолжается бесконечно.

Особое внимание во второй лабораторной работе уделяется эффективному использованию возможностей современного C++. В частности, для хранения списка цветов требуется использовать не просто статический массив в стиле C, а контейнер std::vector из стандартной библиотеки шаблонов stl.



Цели лабораторной работы и порядок её выполнения.

Лабораторная работа №2 строится на основе предыдущей работы с внесением необходимых дополнений. При этом, к основным целям лабораторной работы относятся:

1. **Ознакомление с библиотекой GLM.** Данная библиотека является весьма полезной при программировании трехмерной графики и будет широко использоваться на протяжении всех работ данного курса. Кроме того, она будет использоваться при написании программ в рамках курса «Архитектура графических систем». В связи с этим, необходимо ответственно подойти к изучению основ данной библиотеки.
2. **Ознакомление с контейнером std::vector из stl.** Контейнер std::vector является, пожалуй, самым широко используемым контейнером stl и встречается практически в каждой программе, поэтому жизненно необходимо знать, что из себя представляет данный контейнер, и уметь его использовать в собственных программах.
3. **Усвоение принципов разделения обязанностей.** При проектировании приложений трехмерной графики необходимо понимать, что весь функционал распределяется, как правило, между двумя основными функциями. Первая функция (display) занимается выводом изображения на экран, основываясь не некоторых глобальных данных, например, на списке цветов и индексе текущего цвета. Вторая функция (simulation) занимается модификацией глобальных данных в соответствии с логикой работы программы, прошедшим временем или действиями пользователя. Каждая из функций должна заниматься своим делом и не лезть в область ответственности другой функции, что гарантирует более легкую отладку и меньшее количество логических ошибок и нестыковок.

Для облегчения написания программы лабораторную работу рекомендуется выполнять в соответствии со следующей последовательностью действий:

1. **Разобраться с библиотекой GLM.** Прежде всего, необходимо разобраться с библиотекой glm: скачать её, подключить к проекту, указать используемое пространство имен и попробовать создать несколько переменных с типами данных, объявленными в библиотеке.

   Как и при работе с любой другой библиотекой, не требуется знать все сто процентов возможностей данной библиотеки, но требуется уметь уверенно пользоваться теми инструментами, которые необходимы для реализации конкретного проекта. В данном случае, речь, прежде всего, идет об использовании векторов из трех и четырех компонентов вещественного типа.

   Для лучшего понимания принципов использования библиотеки рекомендуется не просто просмотреть приведенную теоретическую информацию, но и попробовать скомпилировать некоторые примеры.
2. **Разобраться с контейнером vector.** Все, что было ранее сказано по поводу библиотеки glm, относится и к библиотеке vector. При изучении теоретического материала рекомендуется набирать и запускать представленные примеры для лучшего понимания их работы и синтаксиса работы с контейнером vector. Также рекомендуется просмотреть дополнительные видеоматериалы, ссылки на которые приводятся в соответствующем разделе. Изучение основных принципов использования контейнера vector является, пожалуй, наиболее важной частью данной лабораторной и необходимо потратить на неё столько времени, сколько потребуется!
3. **Разобраться с функциями display и simulate.** Прежде чем переходить к реализации задания к лабораторной работе, необходимо уяснить задачи, которые ставятся перед каждой из этих функций, а также важность разделения их обязанностей. Здесь необходимо определить, какие глобальные данные необходимы функции display, чтобы она могла выводить чайник нужного цвета, а также то, как функция simulation будет менять эти данные.
4. **Реализовать задание к лабораторной работе.** В завершении необходимо реализовать поставленную задачу, тщательно проверить правильность её выполнения, похвалить себя за хорошо проделанную работу, выпить чаю и расслабиться.

Использование библиотеки GLM.

GLM (OpenGL Mathematics) — библиотека для OpenGL, предоставляющая программисту на C++ структуры и функции, позволяющие производить вычисления необходимые для алгоритмов трехмерной графики – например, работы с векторами или матрицами. Одна из особенностей GLM состоит в том, что его реализация основана на спецификации языка программирования шейдеров GLSL (OpenGL Shading Language). Таким образом, программисту, знакомому с языком программирования шейдеров GLSL, использование данной библиотеки не доставит проблем. Вместе с тем, даже для тех, кто не знаком с GLSL, разобраться с основными структурами данных и функциями для работы с ними, не составит большого труда. Саму библиотеку можно скачать с сервера кафедры из папки с дополнительными модулями к данному курсу или с официального сайта:

https://glm.g-truc.net/0.9.9/index.html

Ниже приведено описание некоторых особенностей библиотеки, используемых типов данных, наиболее востребованных операций, а также доступных функций. Для лучшего понимания особенностей работы библиотеки рекомендуется самостоятельно набрать приведенные ниже примеры, запустить программу и изучить результат применения указанных функций. Более подробную информацию можно получить из документации, приведенной на официальном сайте:

https://github.com/g-truc/glm/blob/master/manual.md

http://glm.g-truc.net/0.9.9/api/index.html

Подключение библиотеки GLM.

Библиотека GLM поставляется в виде только заголовочных файлов (header-only library), то есть, без файлов реализации (*.lib), поэтому для ее работы достаточно указать пути к соответствующему заголовочному файлу и явным образом подключить его:

```
#include <glm/glm.hpp>
```

Помимо основной функциональности, объявленной в файле <glm/glm.hpp>, библиотека glm обладает дополнительными возможностями, реализованными в отдельных модулях. К таким модулям относятся:

- <glm/gtc/matrix_transform.hpp> - модуль, в котором реализованы операции по вычислению матрицы проекции или матрицы камеры. Этот модуль будет активно использоваться в проекте следующего семестра;
- <glm/gtc/type_ptr.hpp> - модуль для преобразования встроенных типов данных в массивы C++ для более удобного использования этих типов данных при передаче в качестве параметров функциям OpenGL).

Таким образом, в данной лабораторной работе требуется подключить три заголовочных файла из библиотеки GLM:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

При подключении библиотеки GLM, также как и при подключении библиотеки freeglut, в свойствах проекта необходимо указать, где лежат соответствующие заголовочные файлы, то есть указать путь к распакованной папке glm.

Указание пространства имен:

Все функции и типы данных библиотеки имеют названия в соответствии со спецификацией языка GLSL. Так, например, в библиотеке GLM определена функция для вычисления косинуса, которая называется «cos». Вместе с тем, в стандартной библиотеке «C» уже определена функция с точно таким же именем. Для того чтобы избежать возможного конфликта имен, библиотека GLM определяет все свои типы данных и функции в собственном пространстве имен – glm.

Пример доступа к типам данных и функциям с помощью указания пространства имен приведен ниже. В частности, здесь определена новая переменная (вектор из трех компонент вещественного типа), которая равна нормализованному значению другого вектора (нормализация вектора — это преобразование заданного вектора в вектор, указывающий в том же направлении, но имеющий единичную длину):

```
// исходный вектор
glm::vec3 vec1 = vec3 (10.0, 0.0, 0.0);
// нормализованный вектор
glm::vec3 vec2 = glm::normalize(vec1);
// вывод для проверки
cout << vec1.x << " " << vec1.y << " " << vec1.z << endl;
cout << vec2.x << " " << vec2.y << " " << vec2.z << endl;
```

Для того чтобы каждый раз не указывать, к какому пространству имен принадлежит идентификатор, можно непосредственно указать используемое пространство имен. Выполняется это с помощью директивы using namespace, как показано ниже:

```
// используемое пространство имен
using namespace glm;
```

После этого можно использовать все типы данных и функции из пространства имен glm без его явного указания, что существенно облегчает написание и чтение кода:

```
// исходный вектор
vec3 vec1 = vec3 (10.0, 0.0, 0.0);
// нормализованный вектор
vec3 vec2 = normalize(vec1);
// вывод для проверки
cout << vec1.x << " " << vec1.y << " " << vec1.z << endl;
cout << vec2.x << " " << vec2.y << " " << vec2.z << endl;
```

Используемые типы данных, объявление и инициализация переменных:

Библиотека GLM разрабатывалась специально для нужд трехмерной компьютерной графики, поэтому в ней реализуется большой набор типов данных, традиционно используемых в этой предметной области. К таким типам данных относятся – двух, трех и четырехмерные вектора, а также матрицы различной размерности. Ниже представлено объявление нескольких переменных без их инициализации:

```
// вектор из трех компонент вещественного типа
vec3 a;
// вектор из четырех компонент вещественного типа
vec4 b;
// матрица вещественных чисел размером 4 на 4
mat4 m;
```

Инициализация переменных осуществляется с помощью конструкторов, имена которых совпадают с соответствующим типом данных. При этом конструкторы являются перегруженными и могут принимать различные параметры. Например, вектор из четырех компонент может быть построен объединением вектора из трех компонент и дополнительного вещественного числа:

```
// вектор из трех компонент вещественного типа (2, 0, 0)
vec3 a = vec3(2, 0, 0);
// вектор из четырех компонент вещественного типа (2, 0, 0, 1)
vec4 b = vec4(a, 1);
```

Вектора в glm являются обычными структурами данных, поэтому можно получить прямой доступ к любой компоненте вектора. При этом в компьютерной графике вектора традиционно используются для представления различных данных и для удобства каждый вектор представлен в трех нотациях:

1. Вектор геометрических координат (точек или направлений). В этом случае к компонентам вектора принято обращаться (x, y, z, w) - нотация, известная по школьному курсу геометрии;
2. Вектор для представления цвета - компоненты принято называть (r, g, b, a);
3. Вектор для работы с текстурными координатами - используется преимущественно в трехмерной компьютерной графике и такие вектора, по традиции, называются (s, t, p, q).

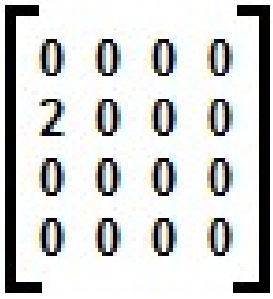
Иными словами, все три нотации являются взаимозаменяемыми и служат исключительно для удобства, чтобы явным образом указать, что имеется в виду, когда происходит работа с вектором. Ниже приводятся несколько примеров по инициализации отдельных компонент вектора. Можно использовать любую нотацию, однако для наглядности желательно использовать ту нотацию, которая больше соответствует назначению переменной:

```
// указание значения каждой компоненты. Итоговый вектор - (2, 0, 0)
vec3 point;
point.x = 2; point.y = 0; point.z = 0;

// другой вариант использования вектора. Итоговый вектор задает красный цвет – (1, 0, 0).
vec3 color;
color.r = 1.0; color.g = 0.0; color.b = 0.0;
```

Отдельного внимания заслуживают матрицы.

Матрицы в компьютерной графике используются, например, для задания положения объектов в составе сцены или для указания положения наблюдателя.



В GLM матрицы представлены двухмерным массивом. Первый индекс указывает номер столбца, второй – номер строки. Столбцы и строки нумеруются с нуля:

```
// Доступ к нулевой колонке, первой строки:
mat4 MT;
MT[0][1] = 2;
```

Основные операции над векторами и матрицами:

В библиотеке GLM для векторов и матриц перегружены основные операции, так что работа с ними носит достаточно очевидный характер. Ниже перечислены основные операции над векторами:

```
// первый вектор
vec3 a = vec3(2, 0, 0);
// второй вектор
vec3 b = vec3(1, 0, 0);
// сложение двух векторов - (3, 0, 0)
vec3 c0 = a + b;
// вычитание двух векторов - (1, 0, 0)
vec3 c1 = a - b;
// умножение вектора на число (скаляр) - (200, 0, 0)
vec3 c2 = 100.0f * a;
// покомпонентное перемножение двух векторов (6, 0, 0)
vec3 c3 = a * c0;
```

Следует обратить внимание, что необходимо явным образом указать, что числовая константа является вещественным числом обычной точности (float), в противном случае тип константы будет определен либо как int (100), либо как double (100.0), что приведет к ошибке компиляции.

Так же одной из наиболее востребованных операций является перемножение матриц или умножение матрицы на вектор. Данная операция используется для преобразования координат, о чем будет рассказано в одной из следующих лабораторных работ. Например, для перевода вершин модели из локальной системы координат в глобальную систему координат, или в систему координат наблюдателя, может использоваться следующий фрагмент кода:

```
// координата вершины модели (x, y, z, w = 1)
vec4 a;
// матрица модели (размещает объект на сцене)
mat4 modelMatrix;
// матрица наблюдателя (указывает позицию наблюдателя)
mat4 viewMatrix;
// координаты вершины в системе координат наблюдателя
vec4 a0 = viewMatrix * modelMatrix * a;
```

Основные функции для работы с векторами и матрицами:

Помимо операций для работы с векторами и матрицами в glm также определен целый ряд функций, которые существенно облегчают реализацию алгоритмов трехмерной графики. Ниже представлены основные функции и их наиболее типичное применение:

1. Нормализация вектора. Результатом работы данной функции является новый вектор, направленный в ту же сторону, что и исходный вектор, но имеющий единичную длину:

```
// вектор
vec3 a;
// нормализованный вектора
a = normalize(a);
```

2. Вычисление длины вектора.

```
// вектор v = (2, 3, 0)
vec3 v = vec3(2, 3, 0);
// длина вектора v = 3.6
float vLength = length(v);
```

3. Определение расстояния между двумя точками:

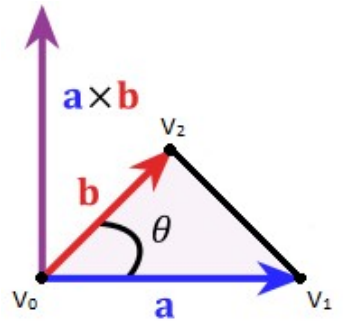
```
// первая точка (0, 5, 0);
vec3 p1 = vec3(0, 5, 0);
// вторая точка (0, 0, 0);
vec3 p2 = vec3(0, 0, 0);
// расстояние между точками = 5.0
float d = distance(p1, p2);
```

4. Вычисление скалярного произведения векторов. Результатом работы функции является значение скалярного произведения двух векторов, которое численно определяется как произведение длин векторов на косинус угла между ними. В случае если оба вектора имели единичную длину (были предварительно нормализованы), скалярное произведение равно значению косинуса угла между векторами:

```
// определение косинуса угла между двумя векторами
vec3 a = vec3(1, 0, 0);
vec3 b = vec3(0, 1, 0);
a = normalize(a);
b = normalize(b);
// вектор a и b взаимно перпендикулярны, поэтому косинус угла между ними равен 0.0
float cos_a_b = dot(a, b);
// получение угла между векторами в радианах (1.570793)
float angleRadians = acos(cos_a_b);
// получение угла между векторами в градусах (90.0)
float angleDegree = degrees(angleRadians);
```

5. Вычисление векторного произведения. Векторное произведение дает новый вектор, перпендикулярный двум предыдущим векторам. Векторное произведение часто используется для определения нормали к поверхности, то есть вектора, перпендикулярного этой поверхности.

   Ниже представлен пример нахождения нормали к полигону, заданному тремя вершинами (V_0, V_1 и V_2) в указанной последовательности. Следует отметить, что для формирования двух векторов (a и b), векторное произведение которых даст необходимый результат, используется вычитание векторов, являющихся вершинами полигона ($a = V_1 - V_0$, $b = V_2 - V_0$). Полученный вектор является перпендикулярным к полигону и направлен «наружу»:




```
// нахождение нормали к треугольнику
vec3 v0, v1, v2;
vec3 normal = cross(v1 - v0, v2 - v0);
```

   Следует отметить, что векторное произведение не является коммутативным, то есть зависит от порядка операндов, поэтому для получения нормали, направленной в указанном направлении, необходим именно такой порядок операций.

6. Нахождение обратной матрицы. Умножение матрицы на вектор часто используется для перевода из одной системы координат в другую. Например, из локальной системы координат в глобальную, используя матрицу модели:

```
// преобразование из локальной системы координат в глобальную
vec4 localCoordinate;
mat4 modelMatrix;
vec4 globaCoordinate = modelMatrix * localCoordinate;
```

   Иногда, однако, необходимо выполнить обратную операцию – переход из глобальной системы координат в локальную. Для такого преобразования необходимо иметь обратную матрицу для матрицы модели. Нахождение обратной матрицы достаточно трудоемкая операция, поэтому glm предлагает готовую функцию для этих целей:

```
// преобразование из глобальной системы координат в локальную
vec4 globaCoordinate;
mat4 modelMatrix;
// нахождение обратной матрицы
mat4 modelMatrixInverse = inverse(modelMatrix);
// получение локальных координат
vec4 localCoordinate = modelMatrixInverse * globaCoordinate;
```

7. Библиотека glm обладает еще множеством полезных функций, информацию о которых можно почерпнуть из официального руководства. Среди них - функции для вычисления синуса или косинуса определенного угла, а также функции для перевода из градусов в радианы и обратно. В случае, если в процессе написания программы возникнет потребность выполнить какую-либо сложную операцию над векторами или матрицами, прежде чем «изобретать велосипед», следует убедиться, что соответствующая функциональность не реализована в самой библиотеке glm. Готовые реализации из библиотеки glm, как правило, являются более быстрыми и надежными, чем то, что может написать неподготовленный программист.

Использование контейнера std::vector.

В программировании часто возникает потребность в хранении большого количества однотипных данных. Если количество данных заранее известно, то можно создать статические или динамические массивы. В случае же, когда количество данных заранее не известно или оно меняется динамически, использование массивов становится затруднительным. Вместо этого, разработчики часто прибегают к использованию специального контейнера из стандартной библиотеки шаблонов C++, который называется вектором.

Вектор (std::vector) – это последовательный контейнер, представляющий массив, размер которого может меняться динамически. В C++ вектор представлен шаблоном класса, благодаря чему он может работать с любыми типами данных, в том числе с классами или структурами. Класс std::vector предоставляет ряд методов, которые позволяют легко управлять содержимым контейнера прозрачным для пользователя способом. Например, при попытке добавить в массив новый элемент метод самостоятельно определит, достаточно ли памяти в текущем массиве или необходимо расширить массив для того, чтобы вместить необходимое количество элементов.

Еще одним преимуществом использование вектора является то, что он самостоятельно занимается выделением и удалением памяти. В этом случае вероятность утечек памяти существенно ниже. В настоящее время разработчики языка рекомендуют использовать именно std::vector, вместо самостоятельного выделения памяти с помощью оператора new.

С вектором связанно такое понятие как вместимость (capacity). Вместимость – это количество элементов, которые можно добавить в контейнер, без необходимости перераспределять память. Внутри класса вектор использует динамически выделенную память для хранения своих элементов. При добавлении нового элемента, если вся ранее выделенная память занята, возникает необходимость выделить новый массив большего размера, скопировать туда старые данные и добавить новый элемент. Данный процесс является достаточно трудоемким, поэтому класс std::vector старается не выделять память каждый раз при добавлении нового элемента. Вместо этого память выделяется большими порциями, рассчитанными сразу на несколько элементов. Это количество элементов и называется текущей вместимостью. Повторное выделение памяти происходит только при исчерпании этого запаса, то есть вместимость изменяется динамически, по мере необходимости. При этом программист, использующий std::vector, может не вдаваться в подробности функционирования контейнера, а сосредоточиться на реализации собственного алгоритма, что существенно ускоряет разработку программного обеспечения.

Таким образом, вектор занимает чуть больше памяти, чем обычный динамический массив, но компенсирует это удобным интерфейсом для добавления или удаления элементов. С целью более рационального выделения памяти, при создании вектора можно указать его вместимость, отсрочив момент перераспределения памяти.

Далее будут перечислены основные этапы работы с std::vector и его наиболее востребованные методы.

Поскольку std::vector является наиболее популярным контейнером при программировании на C++ необходимо особо тщательно подойти к освоению данной темы. Для того, чтобы уверенно пользоваться этим поистине незаменимым инструментом, рекомендуется дополнительно просмотреть два видеоролика, доступных по следующим ссылкам:

https://www.youtube.com/watch?v=1cKvMZOJeeE

https://www.youtube.com/watch?v=jLPqLW2Bp_w

Подключение библиотеки vector.

Для начала работы с контейнером vector, необходимо подключить модуль <vector>. Вектор является частью стандартной библиотеки C++, поэтому нет необходимости в настройках проекта указывать, где лежат заголовочные или библиотечные файлы:

```
#include <vector>
```

Все типы данных модуля <vector>, объявлены в пространстве имен std. Для облегчения оперирования соответствующими типами данных можно указать используемое пространство имен:

```
using namespace std;
```

Создание контейнера vector.

Контейнер vector хранит упорядоченные элементы определенного типа. Это значит, что все элементы контейнера vector лежат друг за другом и к ним можно обращать по их индексу, начинающемуся с нуля. Тип хранимых данных определяется типом, указанным в качестве параметра шаблона класса при создании вектора. Ниже приведен пример создания вектора для хранения элементов типа string. Данный контейнер может быть объявлен следующим образом:

```
// Контейнер для последовательного хранения нескольких строк
vector<string> names;
```

Данный контейнер является пустым, то есть не содержит ни одного элемента. При попытке добавить в него новый элемент будет автоматически выделен новый динамический массив, рассчитанный на несколько элементов. В первый элемент этого массива будет записано указанное значение. Последующие добавляемые элементы также будут записываться в этот массив до тех пор, пока в нём есть место. Как только место закончится, будет выделена новая память большего размера, в неё будут скопированы старые значения, после чего старый динамический массив будет удален. Данная операция осуществляется прозрачным для программиста способом, поэтому о ней можно не беспокоиться. Тем не менее, для более рационального выделения памяти, чтобы отсрочить момент перераспределение памяти, можно установить начальную вместимость контейнера, используя специальный метод reserve():

```
// устанавливаем начальную вместимость (10 элементов)
names.reserve(10);
```

Следует отметить, что функция reserve не ограничивает максимально возможное количество элементов в контейнере, а просто указывает, сколько памяти необходимо зарезервировать для будущего использования. В случае нехватки памяти, она будет автоматически перераспределена для того, чтобы вместить необходимо количество элементов.

Получить текущее количество элементов, а также текущую вместимость контейнера можно с помощью методов size и capacity соответственно:

```
// текущее количество элементов
cout << names.size() << endl;
// текущая вместимость контейнера
cout << names.capacity() << endl;
```

Также необходимо отметить, что при объявлении вектора его можно инициализировать, используя фигурные скобки. В этом случае созданный вектор уже будет иметь какие-то начальные значения, но всегда может быть расширен или сокращен в дальнейшем:

```
// Контейнер для последовательного хранения нескольких строк
vector<string> names{
        "John Snow",
        "Arya Stark",
        "Tyrion Lannister",
        "Daenerys Stormborn of the House Targaryen, First of Her Name, the Unburnt etc."
};
```

Добавление элемента в контейнер.

Наиболее частой операцией, применяемой к контейнеру vector, является операция добавления элемента в конец списка. Для этого, чаще всего, используется метод push_back(). Каждый элемент контейнера имеет свой индекс, начинающийся с нуля. Зная текущий размер контейнера, можно определить индекс элемента, который был только что добавлен, если в этом есть необходимость:

```
// добавление элемента в конец контейнера
names.push_back("Stannis the Mannis");
// получение индекса добавленного элемента (размер минус один)
int index = names.size() - 1;
// вывод индекса на экран для проверки
cout << index << endl;
```

Следует еще раз отметить, что при добавлении нового элемента может произойти перераспределение памяти, поэтому в некоторых случаях для выполнения этой операции требуется чуть больше времени.

Получение доступа к элементам контейнера vector.

Для получения доступа к элементам вектора можно использовать оператор квадратных скобок [], подобно тому, как это происходит с элементами обычного массива. При использовании данного оператора необходимо строго следить, чтобы не выйти за границы массива. Используя квадратные скобки, можно считывать или записывать значение, соответствующее определенному индексу. Ниже приведен пример вывода всех элементов контейнера на экран:

```
// вывод всех элементов на экран
for (size_t i = 0; i < names.size(); ++i) {
	cout << names[i] << endl;
}
```

Для последовательного перебора элементов контейнера можно также использовать иную версию цикла for, так называемый цикл for-each или range-based цикл for. В этом случае в определенную переменную поочередно копируется каждое значение из контейнера. Для удобства тип данных этой переменной, как правило, не указывается и используется ключевое слово auto:

```
// вывод всех элементов на экран, используя range-based цикл for
for (auto name : names) {
	cout << name << endl;
}
```

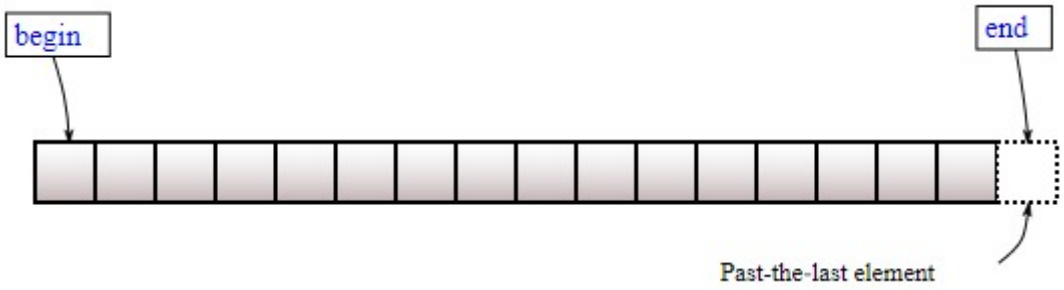
Использование итератора для перебора всех элементов.

Доступ к элементам контейнера vector также может быть выполнен с помощью итераторов. Данный способ используется реже, поскольку доступ к элементам контейнера можно получить более легким способом, используя квадратные скобки []. Тем не менее, итераторы необходимы для удаления элемента из вектора или вставки элемента в конкретную позицию вектора.

Прежде всего, необходимо определить, что такое итератор. Итератор - это класс, который используется для указания на один из элементов контейнера и поддерживает операцию перемещения к следующему элементу данного контейнера. Итератор является шаблоном класса и при его создании необходимо указывать, с каким типом данных работает данный итератор. Каждый контейнер из стандартной библиотеки шаблонов имеет свой собственный тип итератора. В данном случае итератор может быть объявлен следующим образом:

```
// объявление итератора контейнера vector<string>
vector<string>::iterator it;
```

Перед использованием итератор необходимо инициализировать. Класс vector содержит методы begin() и end(), которые возвращают итератор на первый элемент и на элемент «после последнего»:



Итератор не является, как таковым, указателем на элемент, но предоставляет тот же интерфейс для доступа к элементам, что и обычный указатель, в частности: разыменование указателя, переход к следующему элементу, переход к заданному элементу и так далее. Например, вывод второго элемента на экран (индекс 1), может быть выполнен следующим способом:

```
// объявление итератора контейнера vector<string>
vector<string>::iterator it;
// получаем "указатель" на самый первый элемент (индекс 0)
it = names.begin();
// переходим к следующему элементу
it = it + 1;
// выводим элемент на экран (в данном случае "Arya Stark")
cout << (*it) << endl;
```

Часто для сокращения записи используют ключевое слово auto, которое позволяет указать компилятору, что тип данных необходимо определить автоматически. Например, вывод всех элементов контейнера std::vector на экран может быть выполнен следующим способом:

```
// типичный цикл вывода всех элементов контейнера на экран
for (auto it = names.begin(); it != names.end(); ++it) {
	cout << (*it) << endl;
};
cout << endl;
```

Вставка и удаление элементов

Одной из областей, в которой действительно необходим итератор, является выполнение операций вставки элемента в указанную позицию и удаление элемента из указанной позиции. Данные операции выполняются с помощью функций insert и erase соответственно. При этом данные функции принимают в качестве параметра не индекс элемента, а итератор.

Таким образом, процесс вставки или удаления элемента выполняется в два этапа: вначале необходимо создать итератор и передвинуть его в требуемое положение, после чего, используя данный итератор, непосредственно вставить или удалить элемент.

Ниже приводится пример вставки нового элемента в std::vector:

```
// объявление итератора, указывает на элемент с индексом 1 ("Arya Stark")
auto it = names.begin() + 1;
// Добавляем новый элемент.
// Добавление происходит перед элементом, на который указывает итератор
// Все последующие элементы сдвигаются вправо:
// "John Snow", "Ser Bronn of the Blackwater", "Arya Stark", "Tyrion Lannister" и т.д.
names.insert(it, "Ser Bronn of the Blackwater");
```

Удаление проводится аналогичным образом:

```
// объявление итератора, указывает на элемент с индексом 2 ("Arya Stark")
auto it = names.begin() + 2;
// удаляем элемент, на который указывает итератор
// Все элементы смещаются влево на одно значение:
// "John Snow", "Ser Bronn of the Blackwater", Tyrion Lannister" и т.д.
names.erase(it);
```

Операции добавления и удаления, как правило, приводят к перераспределению памяти, то есть выделяется новый массив, в которой копируются все нужные элементы, что существенно сказывается на производительности. В связи с этим, вектор не является оптимальным контейнером для хранения данных, если требуется их частое добавление в середину списка или удаление из середины списка. В этих случаях разумнее использовать другие контейнеры, например, связанный список (std::list).

Важно понимать, что при добавлении элемента в середину списка, или при удалении элемента из середины списка, элементы находящиеся справа смещаются, то есть, в общем случае, индексы элементов будут изменены. То же самое касается и ранее полученных итераторов, которые становятся недействительными. В этом случае необходимо заново получить итератор и передвинуть его в нужную позицию.

Прочие функции контейнера vector.

Контейнер vector содержит некоторые другие полезных функций, часть из которых перечислена ниже:

1. Функция empty – позволяет определить, пуст ли контейнер;
2. Функция size – позволяет узнать, сколько элементов содержится в контейнере;
3. Функция clear – позволяет удалить все элементы из контейнера;
4. Функция shrink_to_fit – позволяет удалить лишнее свободное место, минимизировав потребляемую память, подогнав размер внутреннего динамического массива под действительное количество элементов в векторе.

Также следует упомянуть, что стандартная библиотека шаблонов stl содержит несколько полезных алгоритмов (определенных в модуле <algorithm>, которые эффективным образом работают со стандартными контейнерами. Например, используя функцию sort можно отсортировать std::vector, не тратя время на написание собственных алгоритмов сортировки, которые все равно вряд ли будут эффективнее многократно проверенной реализации стандартной библиотеки:

```
// сортировка списка строк в алфавитном порядке
// указывается итератор начала области и итератор конца области сортировки
// в данном случае сортируется весь список (std::vector)
sort(names.begin(), names.end());
```

Реализация автоматического изменения цвета.

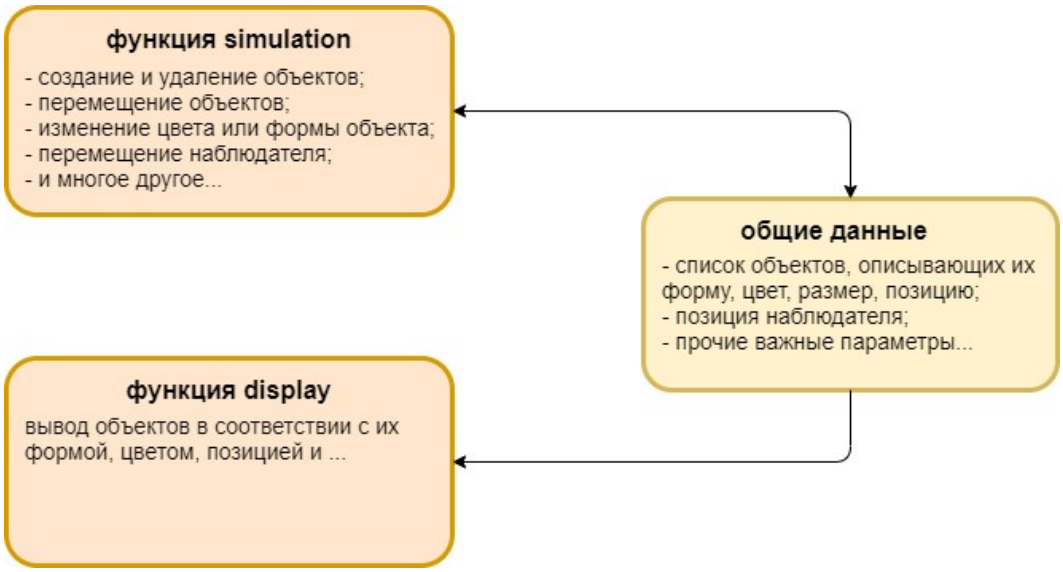
При разработке любых программ важно придерживаться принципа единственной ответственности (single-responsibility principle) в соответствии с которым каждый класс или функция должна иметь одну задачу и быть полностью посвящена её решению. Четкое разделение обязанностей между функциями необходимо для надежной работы программы, легкости её написания, отладки и модернизации.

В программах интерактивной трехмерной графики, в которых требуется реализовать анимацию или сложную логику взаимодействия объектов, функционал программы, как правило, разделяют на две относительно независимые функции:

1. **Функция для вывода объектов на экран (display).** Данная функция должна решать только одну задачу – вывод всех объектов на экран. Для решения задачи функция может обращаться к некоторым глобальным данным. Например, в рамках второй лабораторной работы к таким данным относятся: список цветов, которые могут использоваться для вывода чайника, и индекс, который показывает какой именно цвет из заданного списка используется в текущий момент.

2. **Функция для реализации логики работы программы (simulation).** Эта функция решает задачу «симуляции» - то есть изменения параметров программы в соответствии с предусмотренной моделью поведения. Функция simulation должна вызываться непрерывно и в зависимости от прошедшего времени менять глобальные данные.

Взаимодействие двух функций осуществляется за счет общих данных. При этом структуру данных необходимо тщательно проработать: указать назначение каждой переменной, выбрать подходящий тип данных, дать переменной осмысленно имя и снабдить комментарием. Схематично взаимодействие функции display и simulation может быть проиллюстрировано следующим рисунком:



Двунаправленная стрелка от функции simulation к глобальным данным говорит о том, что функция как считывает эти данным, так и модифицирует их. Например, функция simulation считывает текущую позицию камеры, определяет, нажал ли пользователь клавишу и, при необходимости, передвигает камеру в нужном направлении.

Функция display должна просто считывает данные и использовать их для вывода объектов на экран. Функция не должна менять ни положение камеры, ни положение объектов на экране. Именно поэтому на рисунке от глобальных данных к функции дисплей идет однонаправленная стрелка.

Задание к лабораторной работе.

Лабораторная работа №2 строится на основе предыдущей работы. В рамках лабораторной работы необходимо изучить модуль glm и принципы работы с контейнером std::vector. Кроме того, необходимо:

1. Реализовать автоматическое изменение цвета объекта. Цвет объекта меняется раз в секунду без каких-либо дополнительных действий пользователя в соответствии со следующим списком: белый, синий, красный, желтый и фиолетовый. При достижении последнего цвета из списка происходит возврат к первому цвету и смена цветов продолжается (для лучшего понимания, можно посмотреть пример к лабораторной работе).
2. Для задания списка цветов используется тип vec3 из библиотеки glm, а также контейнер std::vector. Иными словами, список цветов должен быть объявлен следующим образом:

```
// список цветов
vector<vec3> teapotColors;
```

3. Для автоматического изменения цвета необходимо доработать функцию simulation, которая вызывается по таймеру каждые 20 мс. При этом, частоту вызова функции simulation изменять не следует, поскольку чем выше частота симуляции, тем более плавным будет моделируемый процесс. Кроме того, в функции simulation могут осуществляться и другие действия, которые происходят быстрее или медленнее. Для определения того, что прошла одна секунда и цвет необходимо изменить может потребоваться дополнительная переменная.
4. В процессе написания и отладки программы рекомендуется выводить на консоль вспомогательную информацию, чтобы контролировать состояние глобальных переменных, участвующих в симуляции или влияющих на вывод изображения на экран.

Содержание отчета.

1. Титульный лист.
2. Задание к лабораторной работе.
3. Текст программы с комментариями.