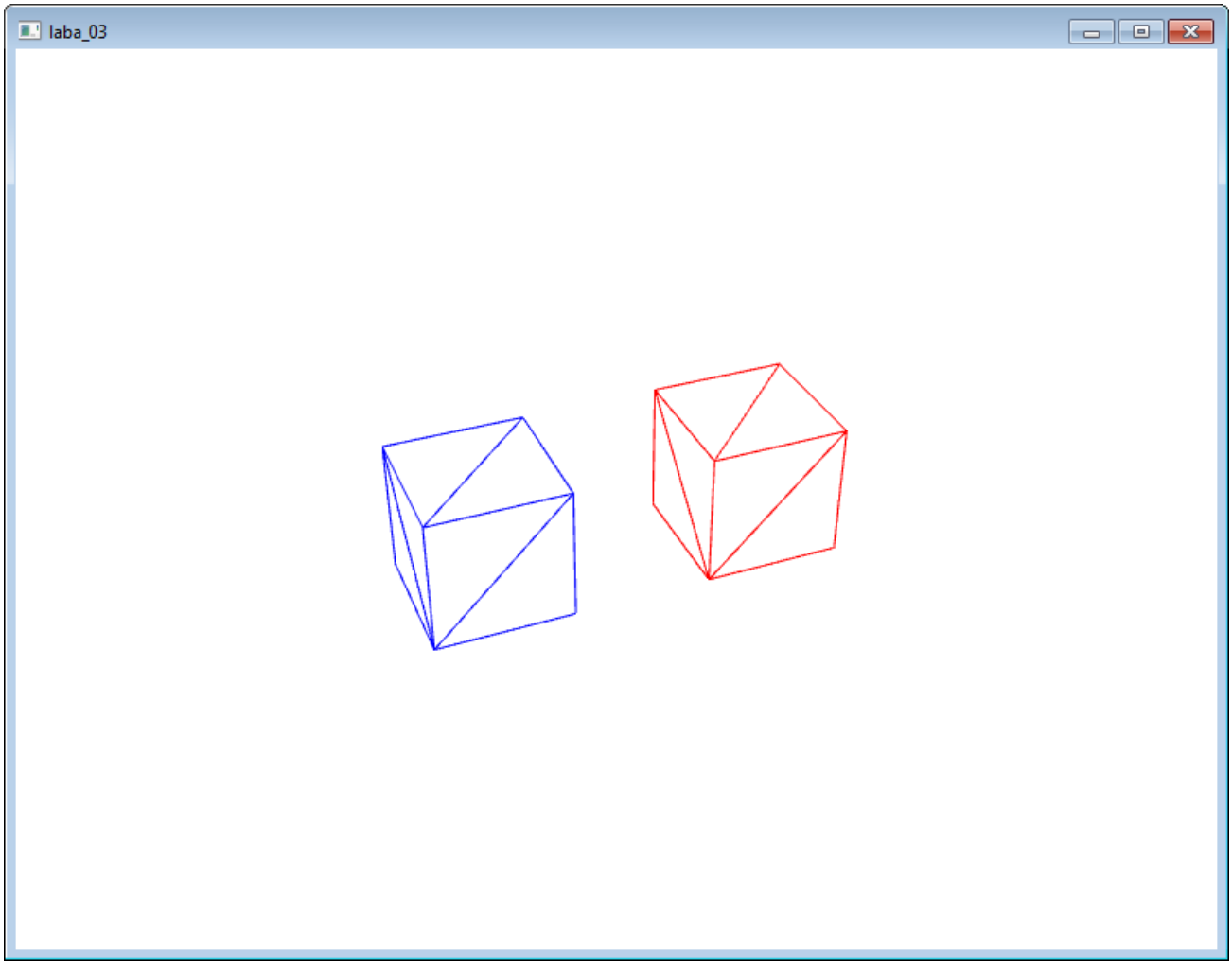
Лабораторная работа №03

Преобразование вершин с использованием матриц наблюдения, модели и проекции в GLSL шейдерах.

Целью лабораторной работы №3 является реализация преобразования вершин с использованием матриц проекции, наблюдения и модели, которые генерируются в основной программе и передаются в GLSL-шейдер через uniform-переменные.



Для закрепления материала и развития навыков программирования необходимо реализовать класс камеры с возможностью задания матрицы проекции и матрицы наблюдения. Камера должна быть направленной и иметь возможность передвигаться и вращаться вокруг точки наблюдения с использованием мышки.

Порядок выполнения лабораторной работы.

Целью данной лабораторной работы является вывод объекта с учетом позиции самого объекта (матрица модели), позиции наблюдателя (матрица наблюдения) и способа проецирования (матрица проекции). Шейдер для реализации указанных преобразований был рассмотрен на лекции и доступен для ознакомления в примере, прилагаемом к данным методическим указаниям.

Наибольшее внимание в данной лабораторной работе уделено генерированию корректных матриц проекций, наблюдения и модели, а так же передачи их в используемый шейдер. В рамках лабораторной работы требуется оформить функциональность камеры в виде отдельного класса. Данная функциональность включает в себя, в том числе, возможность передвигать, вращать и приближать камеру.

Для генерации корректных матриц, а так же для работы с ними, требуется выполнять определенные математические действия, поэтому необходимо задействовать библиотеку GLM (GL Mathematic), рассмотренную в предыдущей лабораторной работе.

Таким образом, лабораторную работу рекомендуется выполнять в следующей последовательности:

1. **Генерация матрицы проекции, матрицы наблюдения и матрицы модели.** Для начала рекомендуется непосредственно сгенерировать матрицу проекции, матрицу наблюдения и матрицу модели, после чего передать необходимые данные в шейдер для вывода модели. Это позволит не только проверить правильность генерирования матриц, но также убедиться в том, что функции для установки uniform-переменных работают корректно.

   Модель должна быть выведена в предсказуемом месте с учетом параметров расположения самой модели и камеры. Для вывода модели необходимо использовать функцию DrawCube, приведенную в приложении 1.

2. **Создание класса для работы с направленной камерой.** После того, как навыки генерирования матриц будут освоены, можно приступать к написанию класса для хранения информации о камере. Класс должен включать не только методы для генерации или получения соответствующих матриц, но также и методы для перемещения камеры. Эти методы необходимо вызывать в соответствующих функциях обратного вызова.

3. **Вывод нескольких объектов на экран.** В завершении необходимо вывести на экран несколько кубиков раскрашенных разным цветом, в разных местах сцены, используя один общий шейдер, но передавая ему разные значения uniform-переменных. На данном этапе не требуется реализовывать классы для представления графического объекта, поскольку данная функциональность будет реализована в следующей лабораторной работе.

Процесс преобразования вершин.

Процесс преобразования вершин, то есть определения того, в каком месте экрана необходимо вывести определенную вершину модели, строится по следующей схеме:

1. Прежде всего, необходимо выбрать шейдер, который будет использоваться для вывода модели на экран. Именно вершинный шейдер определяет алгоритм, который будет использован для вычисления координаты вершины.

2. Далее следует установить значение uniform-переменных в выбранном шейдере. Например, в предыдущей лабораторной работе использовалась переменная “uniform vec2 offset” для вычисления позиции вершины. В рамках данной лабораторной работы для вычисления позиции вершины будут использоваться матрица проекции, матрица наблюдения и матрица модели, которые также являются uniform-переменными.

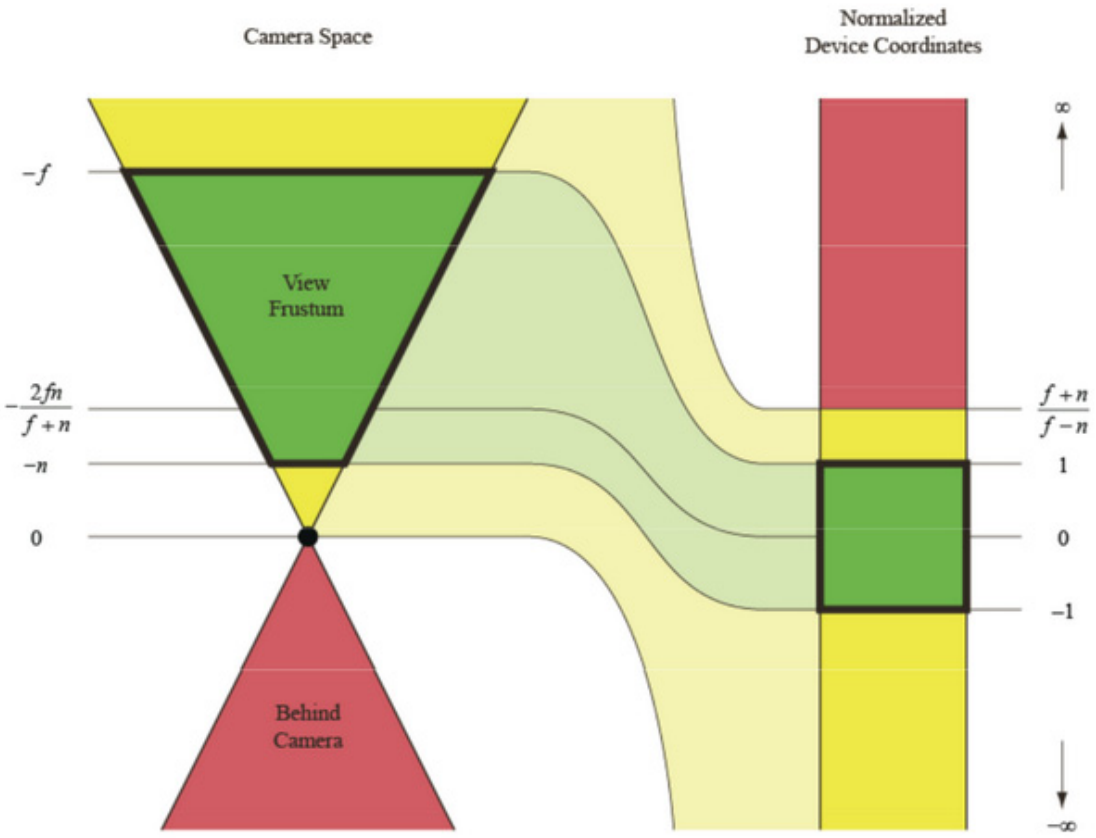
3. Далее следуют непосредственно команды передачи всех вершин модели в OpenGL. Передача вершин подразумевает передачу всех атрибутов каждой вершины в видеокарту. Данные команды будут рассмотрены в следующей лабораторной работе, здесь же для вывода кубика на экран будет использоваться функция DrawCube, полный текст которой приведен в приложении. Данная функция генерирует и передает в OpenGL трехмерные геометрические координаты вершин куба. Сам куб лежит в начале локальной системы координат и имеет длину ребер равную единице, то есть все координаты лежат в пределах от -0.5 до +0.5 по каждой оси.

4. На следующем этапе выполняется вершинный шейдер. Вершинный шейдер, используя uniform-переменные, которые являются общими для всех вершин данной модели, и attribute-переменные, которые являются индивидуальными для каждой вершины, должен вычислить преобразованные координаты вершины и записать их во встроенную переменную gl_Position.
   Переменная gl_Position объявлена как вектор из четырех компонент (x, y, z, w). Важную роль несут не только привычные трехмерные координаты (x, y, z), но и w-компонента, поскольку именно благодаря ей можно добиться эффекта перспективы, при котором объекты, расположенные дальше от наблюдателя, кажутся меньше.
   Вычисленное значение называется усеченной координатой (clip space) и передается дальше по конвейеру рендеринга. На этом работа вершинного шейдера закончена.

5. Далее следует шаг, который называется перспективным делением. Вычисленное значение gl_Position делится на свой компонент w, в результате чего получается вектор (x/w, y/w, z/w, 1). Полученное значение называется нормализованными координатами устройства (normalized device coordinates / NDC). Именно это значение определяет, в каком месте экрана будет лежать вершина. На экране будут видны только те вершины, которые умещаются в «нормализованный куб» - куб в которой каждый компонент меняется в пределах [-1; +1].
   Например, вершина, чья X-координата равны «-1» будет лежать на левой границе области вывода. Вершина, чья x-координата равна «+1» будет лежать на правой границе области вывода. Координата Y определяет положение на нижней («-1») или верхней («+1») области вывода.

Значение Z, хоть и не влияет непосредственно на положение вершины, но используется для принятия решения о видимости/невидимости вершины и связанных с ней фрагментов. В частности, вершина не будет видна вообще, если её Z-координата лежит вне диапазона [-1; +1]. Кроме того, Z-координата так же участвует в определении видимости фрагментов с помощью z-теста.

Наглядный пример соответствия усеченной пирамиды видимости, заданной матрицей проекции, и системой нормализованных координат устройства представлен на следующем рисунке:



6. Последним этапом является трансформация порта просмотра. Трансформация порта просмотра задается с помощью функции glViewport, рассмотренной в прошлом семестре, и указывает, как абстрактные нормализованные координаты устройства преобразуются в реальные пиксели на экране. В частности, данная функция позволяет указать область экрана, в которую будет осуществляться вывод изображения.

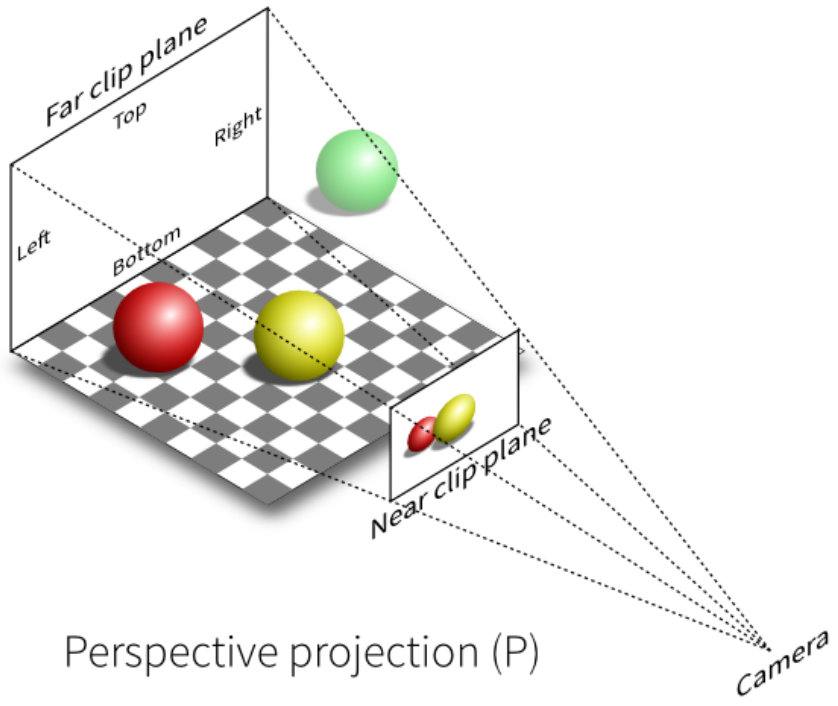
Приведенная выше последовательность действий выполняется всегда. При этом следует обратить внимание, что непосредственно преобразование вершин, то есть вычисление встроенной переменной gl_Position, может осуществляться произвольным образом. На практике, тем не менее, обычно используют стандартный подход, основанный на матрицах. В частности, в рамках данной лабораторной работы, для преобразования вершин необходимо использовать матрицу проекции, матрицу наблюдения и матрицу модели точно так же, как они использовались в фиксированном конвейере рендеринга в лабораторных работах предыдущего семестра.

Использование матриц для размещения объектов сцены.

Традиционно преобразование вершин для вывода моделей на экран выполняется с использованием трех матриц: матрицы модели, матрицы наблюдения и матрицы проекции. Далее каждая из этих матриц будет рассмотрена независимо, а так же будут приведены участки кода, демонстрирующие примеры формирования соответствующих матриц.

Матрица проекции.

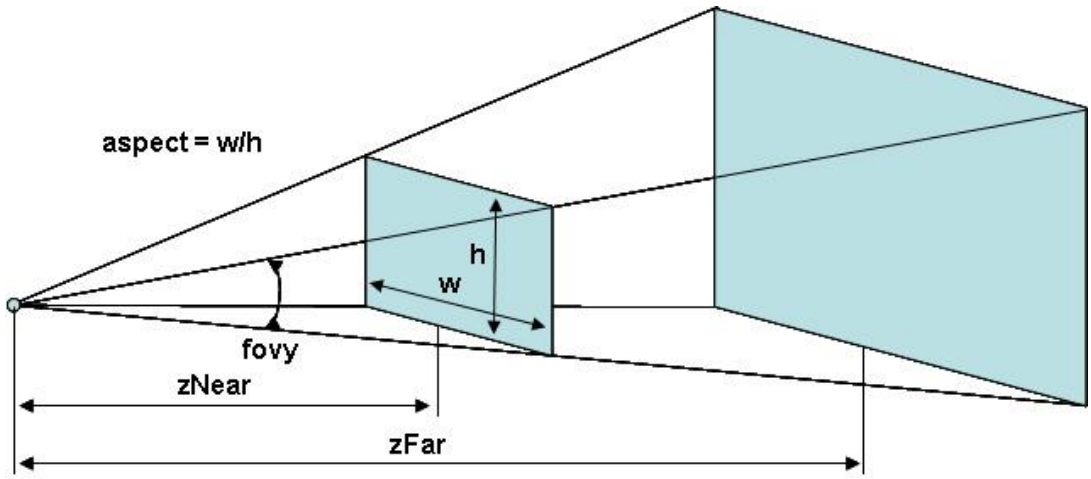
Матрица проекции задает способ проецирования трехмерных координат на экран и одновременно определяет усеченную пирамиду видимости (frustum), которая устанавливает область пространства, видимого наблюдателю. Все что лежит вне усеченной пирамиды видимости не будет выведено на экран. Все что лежит внутри области видимости отображается на экране. В дальнейшем, после выполнения перспективного деления, полученная усеченная пирамида преобразуется к «нормализованному» кубу (кубу, чьи координатами лежат в пределах [-1; +1]). Таким образом, ближняя часть пространства растягивается, и именно поэтому объекты, лежащие дальше от наблюдателя, на экране занимают меньшую площадь при тех же исходных размерах:



В проекте предыдущего семестра, при использовании фиксированного конвейера рендеринга OpenGL, можно было использовать различные встроенные функции, например, glFrustum или gluPerspective для генерации матрицы проекции. В современной версии OpenGL, при использовании шейдеров, такого понятия как текущая матрица проекции или текущая матрица наблюдения модели не существует, поэтому данные функции не поддерживаются, и матрицы необходимо генерировать самостоятельно.

Для удобства генерации стандартных матриц проекции библиотека glm предлагает функцию perspective, которая является аналогом функции gluPerspective. Данная функция имеет четыре параметра, которых достаточно для определения области видимости:

- fovy – угол обзора в вертикальном направлении (в радианах);
- aspect – отношение ширины окна (w) к его высоте (h);
- zNear – ближняя плоскость отсечения;
- zFar – дальняя плоскость отсечения.



Ниже приведен код для генерации матрицы проекции. Данная последовательность инструкций генерирует матрицу проекции так, чтобы описать объем видимости имеющий угол охвата по вертикали в 35 градусов, ближнюю плоскость отсечения равную 1 и дальнюю равную 100. Таким образом, на экран будут выводиться только те объекты, которые находятся от наблюдателя не менее чем в одной и не более чем в 100 единицах. Аспект установлен в вышеуказанное значение, поскольку по умолчанию размер окна в примере равен 800 на 600 пикселей. В случае изменения размеров окна матрица проекции должна быть переопределена заново:

```
// формируем матрицу проекции
mat4   projectionMatrix;
projectionMatrix = perspective(radians(35.0), 800.0 / 600.0, 1.0, 100.0);
```

Отдельно следует отметить важность того, чтобы аспект порта просмотра, устанавливаемый функцией glViewPort, и аспект матрицы проекции совпадали. Это необходимо для того, чтобы избежать искажений объектов, при которых круг будет выведен как эллипс, а квадрат как прямоугольник. Следовательно, матрицу проекции следует переопределять каждый раз, когда меняется размер окна или порт просмотра, то есть в функции обратного вызова Reshape.

Матрица вида (матрица камеры).

Матрица вида необходима для преобразования координат из глобальной системы координат в систему координат наблюдателя (view space). Матрица вида строится с учетом того, где находится наблюдатель, и в какую точку направлен его взгляд. Для облегчения генерирования матрицы наблюдения в библиотеке glm если функция lookAt, которая является аналогом ранее используемой функции gluLookAt. Пример кода для генерации матрицы камеры приведен ниже:

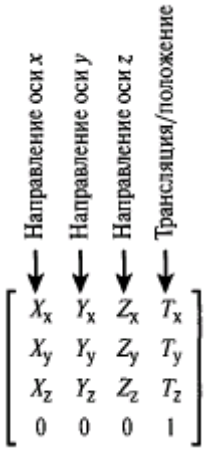
```
// генерирование матрицы камеры
mat4 viewMatrix;
// позиция камеры - (0, 3, 5)
vec3 eye = vec3(0.0, 3.0, 5.0);
// точка, в которую направлена камера - (0, 0, 0);
vec3 center = vec3(0, 0, 0);
// примерный вектор "вверх" (0, 1, 0)
vec3 up = vec3(0, 1, 0);
// матрица камеры
viewMatrix = lookAt(eye, center, up);
```

Следует отметить, что в данной лабораторной работе требуется не только сгенерировать матрицу камеры один раз, но и менять её в зависимости от действий пользователя. Например, пользователь может переместить камеру, повернуть камеру в горизонтальной или вертикальной плоскости или даже приблизить/удалить камеру по отношению к точке наблюдения. Необходимая функциональность должна быть оформлена в виде отдельного класса для работы с камерой, структура которого приведена в соответствующем разделе.

Матрица модели.

Каждая модель, как правило, спроектирована вокруг своей собственной системы координат, которая называется локальной системой координат (системой координат модели, model space). Традиционно считается, что данная система является правосторонней, в которой ось Y направлена вверх.

Для размещения объекта на сцене необходимо указать матрицу модели – матрицу, которая переводит все вершины из локальной системы координат в глобальную систему координат. Для вычисления требуемой матрицы необходимо «разместить» локальную систему координат в определенной точке сцены, выразить оси локальной системы координат и её центр через координаты глобальной системы координат. После чего необходимая матрица вычисляется в соответствии с рисунком приведенном справа.



Следует отметить, что пример генерирования подобной матрицы был подробно рассмотрен в предыдущем семестре. Единственным отличием является то, что теперь используются не стандартные матрицы OpenGL, а пользовательские переменные, которые будут переданы в uniform-переменные, объявленные в шейдере.

Ниже приведен пример матрицы модели для размещения объекта в точке (1, 0, 0) без поворота. Обратите внимание, что для работы с матрицами используется класс mat4 из библиотеки glm. Один из конструкторов данного класса позволяет создать матрицу, передавая в качестве параметров вектора, которые будут записаны в соответствующие столбцы:

```
// матрица модели
mat4 modelMatrix;
// модель располагается в точке (1,0,0) без поворота
modelMatrix = mat4(
	vec4(1, 0, 0, 0),   // 1-ый столбец: направление оси oX
	vec4(0, 1, 0, 0),   // 2-ой столбец: направление оси oY
	vec4(0, 0, 1, 0),   // 3-ий столбец: направление оси oZ
	vec4(1, 0, 0, 1));  // 4-ый столбец: позиция объекта (начала координат)
```

Преобразование координат.

После того, как необходимые матрицы были сгенерированы, их необходимо передать в шейдер в качестве uniform-переменных. При этом усеченные координаты (то есть позиция вершины) будут вычисляться по следующей формуле:

```
gl_Position = projectionMatrix * viewMatrix * modelMAtrix *  vec4 (vPosition, 1);
```

Нетрудно заметить, что для преобразования вершин требуется выполнить три операции матричного умножения, в то время как матрицы модели, проекции и камеры являются неизменные для всех вершин одной модели. В связи с этим, для сокращения количества операций, заранее перемножают матрицу камеры и матрицу модели, получая матрицу наблюдения-модели, которую затем и передают в шейдер. Участок кода для вычисления матриц модели выглядит следующим образом:

```
// устанавливаем матрицу наблюдения модели
mat4 modelViewMatrix = viewMatrix * modelMatrix;
```

Формула преобразования вершины в шейдере заменяется на следующую конструкцию:

```
gl_Position = projectionMatrix * modelViewMatrix * vec4 (vPosition, 1);
```

Таким образом, количество операций умножений для каждой вершины сократилось с трех до двух, что дает существенный выигрыш в производительности. Закономерно возникает вопрос, почему не перемножить сразу и матрицу проекции, передав в шейдер только одну матрицу. Однако следует учесть, что в результате применения матрицы проекции искажаются пропорции объектов и углы между ними (за счет этого достигается эффект перспективы). В связи с этим, если над вершинами в шейдере требуется выполнять еще какие-либо действия, рекомендуется отдельно передавать матрицу наблюдения-модели и отдельно матрицу проекции. К таким действиям, которые требуют сохранения пропорций объектов, в частности, относится вычисление освещения объектов, которое будет рассмотрено в одной из следующих лекций.

Класс для представления камеры.

В рамках данной лабораторной работы требуется реализовать класс для хранения параметров направленной камеры и управления ею. Класс предназначен для решения двух основных задач:

1. Хранения параметров камеры и соответствующих матриц. Класс должен иметь методы для удобного задания матриц проекции и наблюдения, а так же методы для получения соответствующих матриц.
2. Управления камерой. Класс должен иметь методы, которые позволяют в удобной форме управлять камерой, в частности: передвигать камеру в горизонтальной плоскости, поворачивать её вправо/влево и вверх/вниз, а так же приближать и удалять камеру по отношению к точке наблюдения. При изменении позиции камеры матрица наблюдения должна быть пересчитана. Пример передвижения камеры приведен в демонстрационной программе, прилагаемой к данным методическим указаниям.

В данной лабораторной работе нужно не только организовать вывод кубиков с учетом способа проецирования и позиции наблюдателя, но, так же, реализовать дополнительные возможности, такие как, например, перемещение камеры. Полный список требований выглядит следующим образом:

1. Камера должна быть направленной, то есть у камеры есть текущая позиция и точка, в которую она смотрит. Точка наблюдения не обязательно совпадает с началом системы координат, то есть не обязательно равна (0, 0, 0) и по ходу работы программы может меняться.
2. По нажатию на клавиши управления курсором (стрелки вверх/вниз/влево/вправо) происходит передвижение камеры в горизонтальной плоскости. Изменяется как точка наблюдения, так и позиция камеры. Передвижение происходит вперед/назад или вправо/влево с учетом текущей ориентации камеры «естественным» способом. Передвижения камеры осуществляются с постоянной скоростью, не зависящей от производительности компьютера. То есть, при передвижении камеры необходимо отслеживать, сколько прошло времени с момента последнего обновления позиции камеры. Для этого рекомендуется использовать ранее изученные функции QueryPerformanceCounter и QueryPerformanceFrequency.
3. Вращение камеры происходит вокруг точки наблюдения при зажатой правой кнопке мыши и перемещении курсора. Скорость поворота определяется с учетом того, насколько сильно изменилось положение курсора мышки. Состояние клавиш, в том числе и состояние правой кнопки мыши, можно получить, используя функцию GetAsyncKeyState, а текущую позицию курсора мышки - с помощью функции GetCursorPos из библиотеки WinAPI. При вращении в вертикальной плоскости предусмотреть предельные углы в 5-85 градусов.

Как и любую другую относительно независимую сущность, требующую обработки, камеру необходимо организовать как отдельный класс и поместить в собственный модуль. Ниже приведен рекомендуемый состав класса, однако разрешается вносить любые оправданные изменения в его структуру. При этом в классе умышленно не приводятся скрытые поля класса, необходимые для его работы, а приводятся только интерфейсные функции. Поля класса, такие как, например, текущие значения матриц проекции и наблюдения, текущая позиция камеры, точка наблюдения и прочие параметры требуется определить самостоятельно.

```
// КЛАСС ДЛЯ РАБОТЫ С КАМЕРОЙ
class Camera
{
	// внутренние переменные, требуемые для представления камеры
	...
public:
	// конструктор по умолчанию
	Camera (void);

	// установка матрицы проекции
	void setProjectionMatrix(float fovy, float aspect, float zNear, float zFar);
	// получение матрицы проекции
	mat4 getProjectionMatrix();

	// получение матрицы наблюдения
	mat4 getViewMatrix();

	// передвинуть камеру и точку наблюдения в горизонтальной плоскости oXZ
	void move(float dx, float dz);
	// вращение в горизонтальной и вертикальной плоскости
	void rotate(float horizAngle, float vertAngle);
	// приближение/удаление
	void zoom(float dR);
};
```

Следует обратить внимание на отсутствие метода apply, который использовался в предыдущем проекте для передачи параметров камеры в OpenGL. Это связанно с тем, что при использовании шейдеров нет такого понятия как текущая матрица проекции или матрица наблюдения модели, а указанные параметры должны передаваться в каждый используемый шейдер.

Для взаимодействия с пользователем, в частности, для вызова определенных методов камеры при нажатии клавиш, перемещения курсора мышки или вращения её колесика, используется система функций обратного вызова, которые приведены в следующем разделе.

Использование функций обратного вызова.

Для взаимодействия с пользователем в данных лабораторных работах используется система функций обратного вызова (callback functions). Функции обратного вызова, это функции которые регистрируются в системе (в данном случае, в библиотеке glut) и которые будут вызваны автоматически при наступлении какого-либо события для его обработки. Для регистрации функций обратного вызова, то есть для указания того, какие функции и в каком случае должны быть вызваны, используются специальные методы библиотеки freeglut. Рассмотрим основные функции обратного вызова и их применение.

Функция Reshape. Функция Reshape регистрируется с помощью функции glutReshapeFunc и будет вызываться автоматически каждый раз при изменении размеров окна. Функция вызывается, в том числе, и при создании окна. Основное назначение функции Reshape – это изменение порта просмотра и матрицы проекции с учетом новых размеров окна. Полный код функции выглядит следующим образом:

```
// функция, вызываемая при изменении размеров окна
void Reshape (int w,int h)
{
	// установить новую область просмотра, равную всей области окна
	glViewport(0,0,(GLsizei)w, (GLsizei)h);
	// устанавливаем матрицу проекции
	camera.setProjectionMatrix(35.0f, (float)w / h, 1.0f, 100.0f);
};
```

Функция Simulation. Функция **р**егистрируется с помощью функции glutIdleFunc и будет вызываться всякий раз при простое процессора, то есть максимально часто. В данной функции необходимо определить время симуляции, то есть, сколько прошло времени с момента последнего вызова данной функции. Данное время используется для управления скоростью передвижения камеры. Время симуляции определяется с помощью пары функций QueryPerformanceCounter и QueryPerformanceFrequency так же, как это выполнялось в предыдущем проекте.

Кроме того, в данной функции необходимо определить, требуется ли передвигать или поворачивать камеру. Состояние клавиш, в том числе и состояние правой кнопки мыши, можно получить, используя функцию GetAsyncKeyState. Текущую позицию курсора мышки можно получить с помощью функции GetCursorPos.

Общая структура функции выглядит следующим образом:

```
// функция вызывается когда процессор простаивает, т.е. максимально часто
void Simulation(void)
{
	// определение времени симуляции
	…

	// определяем необходимость передвижения камеры
	…
	camera.move(dx, dz);

	// определяем необходимость вращения камеры
	…
	camera.rotate(rx, ry);

	// вызываем принудительную перерисовку окна
	glutPostRedisplay();
};
```

Функция Display. Данная функция регистрируется с помощью функции glutDisplayFunc и вызывается всякий раз, когда требуется перерисовать окно. В том числе при создании окна, изменении его размеров или после использования функции glutPostRedisplay. Функция должна отчистить окно, вывести необходимые объекты, используя определенные шейдеры, камеры и прочие параметры, после чего поменять местами передний и задний буфер. Полный текст функции представлен ниже:

```
// функция вызывается при перерисовке окна
// в том числе и принудительно, по командам glutPostRedisplay
void Display (void)
{
	// отчищаем буфер цвета
	glClearColor (1.0, 1.0, 1.0, 1.0);
	glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
	glEnable (GL_DEPTH_TEST);
	glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
	glEnable(GL_CULL_FACE);
	glCullFace(GL_BACK);

	// активируем шейдер
	shader.activate();

	// устанавливаем матрицу проекции
	mat4 projectionMatrix = camera.getProjectionMatrix();
	shader.setUniform("projectionMatrix", projectionMatrix);

	// получаем матрицу вида (камеры)
	mat4 viewMatrix = camera.getViewMatrix();

	// устанавливаем матрицу наблюдения-модели для 1-го объекта
	mat4 modelMatrix = mat4(
		vec4(1, 0, 0, 0),		// 1-ый столбец: направление оси ox
		vec4(0, 1, 0, 0),		// 2-ой столбец: направление оси oy
		vec4(0, 0, 1, 0),		// 3-ий столбец: направление оси oz
		vec4(-1, 0, 0, 1));		// 4-ый столбец: позиция объекта (начала координат)
	shader.setUniform("modelViewMatrix", viewMatrix * modelMatrix);

	// выводим первый объект красного цвета
	shader.setUniform("color", vec4(1.0, 0.0, 0.0, 1.0));
	DrawCube();

	// устанавливаем матрицу наблюдения-модели для 2-го объекта
	modelMatrix = mat4(
		vec4(1, 0, 0, 0),		// 1-ый столбец: направление оси ox
		vec4(0, 1, 0, 0),		// 2-ой столбец: направление оси oy
		vec4(0, 0, 1, 0),		// 3-ий столбец: направление оси oz
		vec4(1, 0, 0, 1));		// 4-ый столбец: позиция объекта (начала координат)
	shader.setUniform("modelViewMatrix", viewMatrix * modelMatrix);

	// выводим второй объект синего цвета
	shader.setUniform("color", vec4(0.0, 0.0, 1.0, 1.0));
	DrawCube();

	// смена переднего и заднего буферов
	glutSwapBuffers();
};
```

Следует обратить внимание, что шейдеры замещают собой только некоторые ступени конвейера, а не весь конвейер OpenGL. В частности, как видно из приведенного примера, по-прежнему работают такие режимы OpenGL как отсечение лицевых/нелицевых граней и режим вывода полигонов (сплошная заливка или в виде линий).

Функция MouseWheel. Последняя функция обратного вызова, используемая в данной лабораторной работе, ранее не использовалась и необходима для отслеживания вращения колесика мышки. Данная функция регистрируется с помощью функции glutMouseWheelFunc.

Общая структура функции приведена ниже:

```
// функция обработки вращения колесика мышки
void MouseWheel(int wheel, int direction, int x, int y)
{
	// определяем, на сколько необходимо приблизить/удалить камеру
	float delta = ...;
	Camera.Zoom(delta);
}
```

Задание к лабораторной работе.

В лабораторной работе №3 необходимо:

1. В отдельном модуле реализовать класс для работы с камерой. Класс должен включать методы для получения матриц проекции и наблюдения, а так же методы для перемещения камеры. Перемещение камеры должно осуществляться по вышеописанным правилам и в соответствии с приведенным примером.
2. Реализовать вывод нескольких (не менее 7) кубиков расположенных в разных точках сцены, каждый кубик описывается своим собственным цветом и матрицей модели. Реализовывать отдельные классы или структуры для представления объектов на данном этапе не требуется.

Содержание отчета.

1. Титульный лист.
2. Задание к лабораторной работе.
3. Текст класса для работы с камерой (как *.h так и *.cpp файл).
4. Текст функции simulate.

Критерии оценки и вопросы к защите.

Вопросы по теоретической части:

1.1 Какие логические матрицы используются для преобразования вершин?

1.2 Что такое перспективное деление и для чего оно используется?

1.3 Что такое нормализованные координаты устройства?

1.4 Как происходит установка uniform-переменных?

Вопросы по практической части:

1.1 Приведите назначение параметров функции glm::perspective.

1.2 Приведите назначение параметров функции glm::lookAt.

1.3 Приведите назначение параметров функции glUniformMatrix4fv.

Требования к программе:

1.1 Класс для работы с камерой оформлен в виде отдельного модуля.

1.2 Методы для установки uniform-переменных должны использовать ассоциативный контейнер.

Приложение 1. Процедура инициализации и вывода модели.

В качестве модели в данной лабораторной работе используется кубик, лежащий в центре локальной системы координат и имеющий длину ребер равную единице. Для вывода кубика используется функция DrawCube, приведенная ниже. Все необходимые данные объявлены в самой функции как статичные переменные, которые инициализируются в момент первого вызова данной функции:

```
// функция вывода кубика с ребрами единичной длины (-0,5 .. +0,5)
void DrawCube()
{
	// переменные для вывода объекта (прямоугольника из двух треугольников)
	static GLuint VAO_Index = 0;	// индекс VAO-буфера
	static GLuint VBO_Index = 0;	// индекс VBO-буфера
	static int VertexCount = 0;	// количество вершин
	static bool init = true;

	if (init) {
		// создание и заполнение VBO
		glGenBuffers(1, &VBO_Index);
		glBindBuffer(GL_ARRAY_BUFFER, VBO_Index);
		Glfloat Verteces[] = {
			// передняя грань
			-0.5,  +0.5,  +0.5,
			-0.5,  -0.5,  +0.5,
			+0.5,  +0.5,  +0.5,
			+0.5,  +0.5,  +0.5,
			-0.5,  -0.5,  +0.5,
			+0.5,  -0.5,  +0.5,
			// задняя грань
			+0.5,  +0.5,  -0.5,
			+0.5,  -0.5,  -0.5,
			-0.5,  +0.5,  -0.5,
			-0.5,  +0.5,  -0.5,
			+0.5,  -0.5,  -0.5,
			-0.5,  -0.5,  -0.5,
			// правая грань
			+0.5,  -0.5,  +0.5,
			+0.5,  -0.5,  -0.5,
			+0.5,  +0.5,  +0.5,
			+0.5,  +0.5,  +0.5,
			+0.5,  -0.5,  -0.5,
			+0.5,  +0.5,  -0.5,
			// левая грань
			-0.5,  +0.5,  +0.5,
			-0.5,  +0.5,  -0.5,
			-0.5,  -0.5,  +0.5,
			-0.5,  -0.5,  +0.5,
			-0.5,  +0.5,  -0.5,
			-0.5,  -0.5,  -0.5,
			// верхняя грань
			-0.5,  +0.5,  -0.5,
			-0.5,  +0.5,  +0.5,
			+0.5,  +0.5,  -0.5,
			+0.5,  +0.5,  -0.5,
			-0.5,  +0.5,  +0.5,
			+0.5,  +0.5,  +0.5,
			// нижняя грань
			-0.5,  -0.5,  +0.5,
			-0.5,  -0.5,  -0.5,
			+0.5,  -0.5,  +0.5,
			+0.5,  -0.5,  +0.5,
			-0.5,  -0.5,  -0.5,
			+0.5,  -0.5,  -0.5
		};
```

```
            glBufferData(GL_ARRAY_BUFFER, sizeof(Verteces), Verteces, GL_STATIC_DRAW);

            // создание VAO
            glGenVertexArrays(1, &VAO_Index);
            glBindVertexArray(VAO_Index);
            // заполнение VAO
            glBindBuffer(GL_ARRAY_BUFFER, VBO_Index);
            int location = 0;
            glVertexAttribPointer(location, 3, GL_FLOAT, GL_FALSE, 0, 0);
            glEnableVertexAttribArray(location);
            // "отвязка" буфера VAO на всякий случай, чтоб случайно не испортить
            glBindVertexArray(0);

            // указание количество вершин
            VertexCount = 6 * 6;
            init = false;
      }
      glBindVertexArray(VAO_Index);
      glDrawArrays(GL_TRIANGLES, 0, VertexCount);
}
```

Приложение 2. Текст шейдера для вывода модели.

Для вывода модели на данном этапе используется стандартный подход к преобразованию вершин и шейдер, приведенный ниже:

Вершинный шейдер:

```
#version 330 core

uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;
uniform vec4 color;

in vec3 vPosition;

void main ()
{
      gl_Position = projectionMatrix * modelViewMatrix * vec4 (vPosition, 1);
}
```

Фрагментный шейдер:

```
#version 330 core

uniform vec4 color;

out vec4 outputColor;

void main (void)
{
      outputColor = color;
}
```