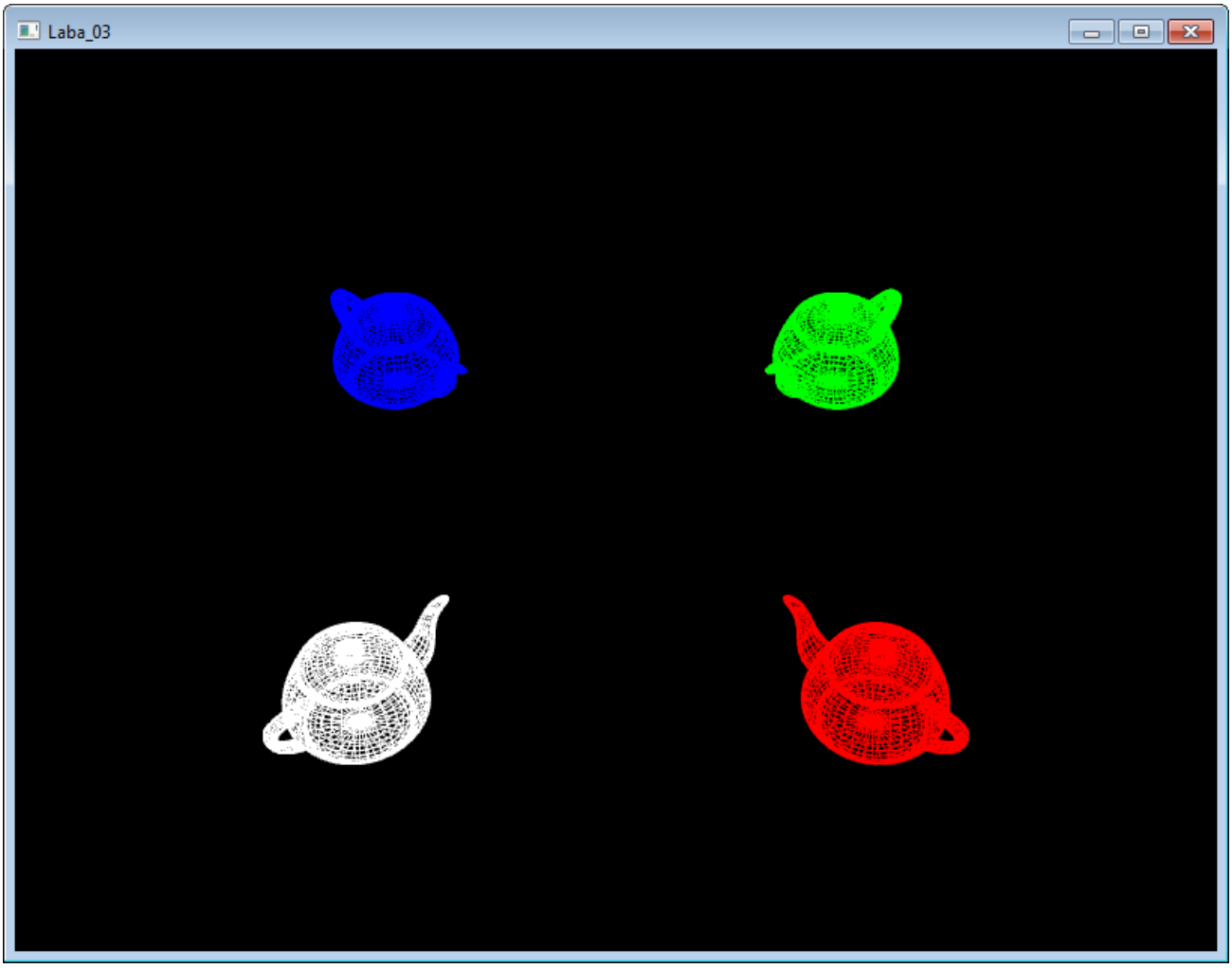
Лабораторная работа №3

Задание положения камеры (наблюдателя).

В данной лабораторной работе необходимо реализовать движение направленной камеры при нажатии на определенные клавиши. Камера реализуется как отдельный класс в собственном модуле с необходимыми полями и методами.



Для более полного использования возможностей, предоставляемых операционной системой, необходимо ознакомится с некоторыми функциями WinAPI. В частности, рассматриваемые здесь функции WinAPI будут использоваться для определения нажатия клавиш и для вычисления времени, прошедшего между двумя последовательными вызовами функции Simulation, то есть для задания скорости движения камеры.

Порядок выполнения лабораторной работы.

В данной лабораторной работе необходимо реализовать класс для работы с направленной камерой. Основным требованием является реализация различных поворотов камеры со строго определенной скоростью. Например, поворот камеры должен происходить при нажатии на клавиши перемещения курсора (стрелки) с такой скоростью, что полный оборот в горизонтальной плоскости должен происходить за 4 секунды, независимо от производительности компьютера. Поскольку положение камеры меняется постепенно в течение нескольких кадров, необходимо определить, на сколько градусов следует повернуть камеру для каждого конкретного кадра. Для этого необходимо знать, сколько времени прошло между кадрами. К сожалению, функция glutTimerFunc не обеспечивает ни необходимой точности, ни гибкости управления. Поэтому для точного определения времени, прошедшего между кадрами, будут использоваться специальные функции WinAPI. Так же функция WinAPI будет использоваться для определения того, какие клавиши нажаты в каждый конкретный момент.

При планировании выполнения лабораторной работы важно определить не только основные этапы, но и способы отладки этих этапов. Здесь предлагается следующий порядок выполнения лабораторной работы:

1. Функцию simulation необходимо вызывать не по таймеру, с заранее строго определенными интервалами, а максимально часто, для чего её необходимо зарагестрировать в качестве функции обратного вызова с помощью функции glutIdleFunc.

2. Разобраться с функциями QueryPerformanceCounter и QueryPerformanceFrequency, которые позволяют определить точные временные промежутки, прошедшие между различными событиями. Функции весьма полезны и используются во многих реальных приложениях для вычисления прошедшего времени или формирования задержек. Необходимо не только разобраться с этими функциями, но и убедиться, что их использование реализовано корректно.

3. Разобраться с функцией GetAsyncKeyState для считывания состояния каждой клавиши. Работа функции достаточно очевидна, и правильность ее вызова легко проверить. Для этого можно, например, поставить точку останова или выводить на консоль информацию о результате работы данной функции.

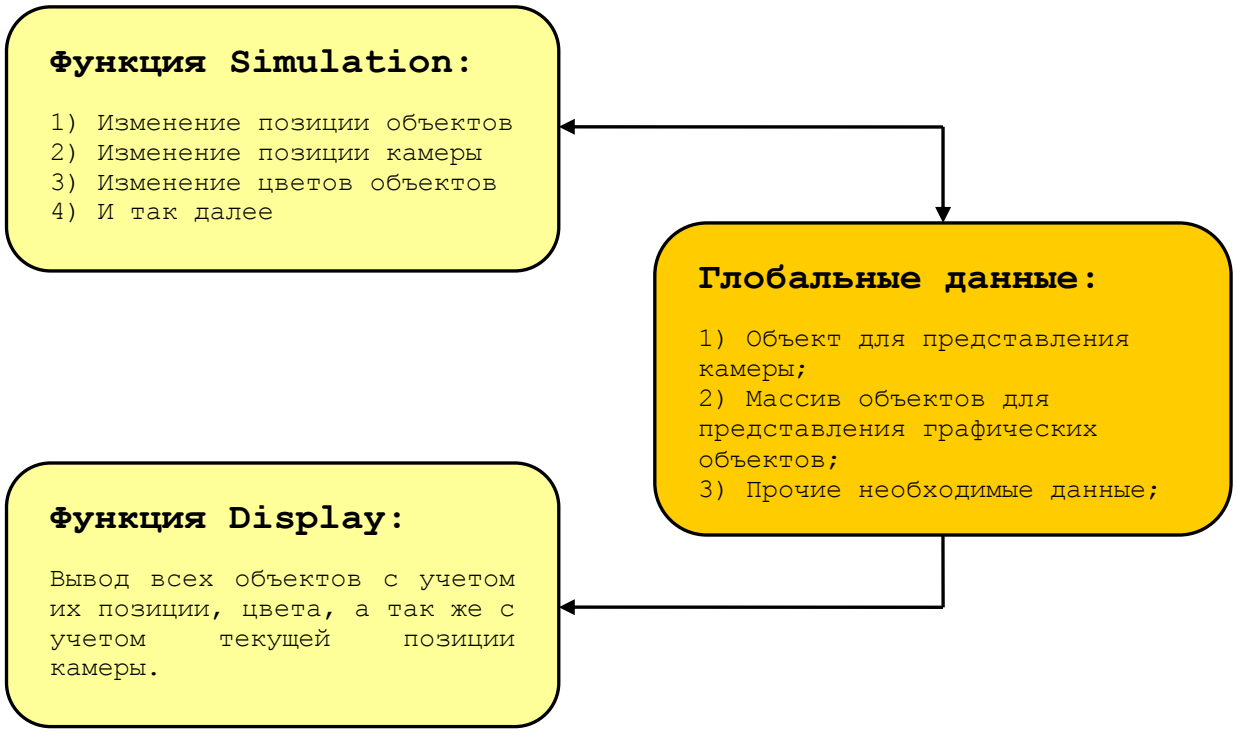
4. В самом конце можно переходить к написанию класса для реализации направленной камеры. Для упрощения задачу лучше решать последовательно. Например, вначале реализовать наиболее важный метод класса – метод для установки матрицы камеры, даже если пока будут использоваться параметры по умолчанию (те же, что и в предыдущей работе). Затем необходимо реализовать движение камеры с учетом нажатых клавиш.

Разделение программы на функции.

В рамках данной лабораторной работы требуется вывести на экран четыре трехмерных объекта (так же как и в прошлой лабораторной работе), а так же реализовать передвижение камеры. Иными словами, необходимо не только выводить на экран определенные объекты, но и менять исходные данные, такие как положение камеры или объектов. При этом изменение данных должно происходить с определенной скоростью.

В программах интерактивной трехмерной графики, в которых требуется реализовать анимацию или сложную логику взаимодействия объектов (например, изменения позиции камеры или объектов), как правило, весь функционал разделяют на две относительно независимые функции: функцию для вывода объектов на экран (Display), и функцию для обработки логики работы программы (Simulation). Четкое разделение обязанностей между двумя функциями необходимо для надежной работы программы, а так же легкости её написания, отладки и модернизации.

Взаимодействие двух функций осуществляется за счет общих глобальных данных, таких как: объекта для представления камеры, массива графических объектов и прочих необходимых данных. Схематично такое взаимодействие 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 только считывает данные для вывода объектов на экран и не должна менять ни положение камеры, ни положение объектов на экране. Именно поэтому на рисунке от глобальных данных к функции дисплей идет однонаправленная стрелка.

Функция вывода объектов на экран (Display).

Данная функция предназначена для вывода всех необходимых объектов на экран в соответствии с их параметрами, такими как цвет или позиция. В процессе своей работы функция обращается к глобальным данным, например, массиву графических объектов или камере. На основе этих данных строится изображение на экране. Функция ни в коем случае не должна модифицировать глобальные данные (не должна передвигать объекты или камеру), поскольку это не входит в ее компетенцию.

Функция Display может быть достаточно сложной, поскольку для построения фотореалистичного изображения нужно применяется множество различных приемов и алгоритмов. Это является еще одной причиной выделения данного функционала в отедльную функцию. Сама функция вызывается в двух случаях:

1. Необходимо перерисовать окно. Данная ситуация возникает, например, в случае изменения его размеров или перекрытии части окна другими окнами. В этом случае операционная система посылает приложению определенное сообщение, которое говорит о том, что окно необходимо отрисовать еще раз. Данное сообщение, обрабатывается библиотекой freeglut, которая вызывает функцию Display, поскольку именно эта функция была ранее зарегистрирована с помощью функции glutDisplayFunc. При этом функция Display просто заново рисует все те же самые объекты в том же самом месте.
2. Произошло изменение глобальных данных, которые влияют на результирующее изображения, то есть передвинулись объекты, камера или изменились еще какие-то параметры. В этом случае необходимо либо самостоятельно вызвать эту функцию, либо с помощью функции glutPostRedisplay уведомить библиотеку о том, что требуется перерисовка. В результате объекты будут выведены на экран с измененными параметрами.

Ниже приведен полный код функции Display. Примечательно, что за счет разбиение функциональности на классы, а так же благодаря осмысленному заданию имен переменных и методов, логика работы функции очевидна. Кроме того, это позволяет произвести изменение способов представления графических объектов или камеры, без внесения изменения в саму функцию Display:

```
// функция вызывается при перерисовке окна
// в том числе и принудительно, по командам glutPostRedisplay
void Display(void)
{
	// отчищаем буфер цвета
	glClearColor(0.00, 0.00, 0.00, 1.0);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	// включаем тест глубины
	glEnable(GL_DEPTH_TEST);

	// устанавливаем камеру
	camera.apply();

	// выводим чайники
	for (int i = 0; i < 4; i++) {
		graphicObjects[i].draw();
	};

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

Функция реализации логики работы программы (Simulation).

Функция Simulation непосредственно реализует логику программы, то есть реализует взаимодействия объектов, изменение их позиций по выбранным алгоритмам, изменение расположения камеры и так далее. Данная функция предназначена просто для изменения некоторых глобальных данных, которые в дальнейшем будут использоваться функцией Display для вывода всей сцены на экран.

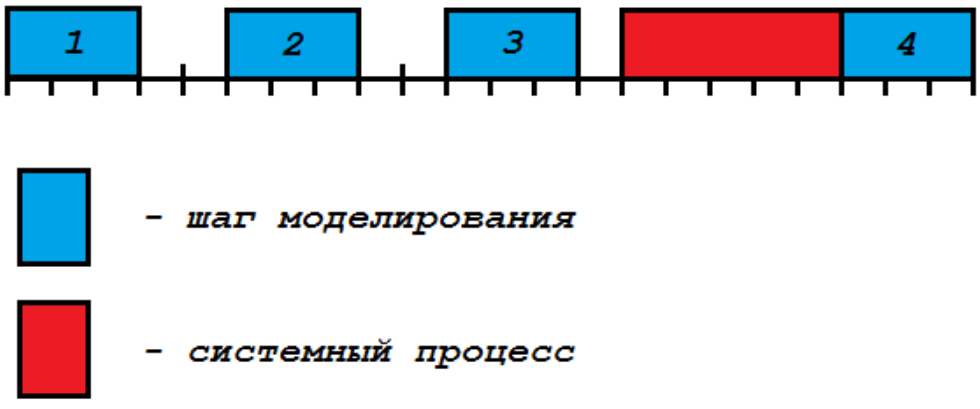
Процесс изменения параметров объектов в интерактивных графических приложениях называется моделированием или симуляцией и выполняется по шагам. Для этого сначала вычисляется один шаг, после чего изображение выводится на экран. Затем вычисляется второй шаг, объекты чуть-чуть меняют свои параметры, например, сдвигаются в определенном направлении, и изображение снова выводится на экран и так далее. На каждом шаге симуляции важно знать, сколько времени прошло с предыдущего шага, для того чтобы правильно смоделировать все процессы, например, определить на сколько нужно переместить объекты или камеру, насколько пополнилась манна персонажа, успело ли перезарядится оружие и прочие, чрезвычайно важные вещи.

При этом существуют два подхода. В первом случае функция моделирования непрерывно вызывается с определенной частотой, скажем, каждые 20 миллисекунд (50 раз в секунду), что было реализовано в первых двух лабораторных работах. В этом случае можно рассчитывать, что между двумя последовательными шагами моделирования прошло ровно 20 мс и, соответственно, зная текущую скорость объекта, можно передвинуть его на определенное расстояние. Такой подход, однако, имеет два существенных недостатка:

1. При указании слишком большого интервала времени между кадрами, процессор некоторую часть времени будет простаивать, что является неэффективным использованием процессорного времени. Кроме того, чем меньше кадров в секунду формирует программа, тем более прерывистым получается изображение;

2. Второй недостаток заключается в том, что всегда существует вероятность того, что в момент, когда необходимо вызвать функцию simulation, операционная система будет занята решением своих системных задач. То есть вызов функции simulation будет отложен на неопределенное время, что приведет к тому, что объекты будут передвигаться с неравномерной скоростью.

Временная диаграмма, иллюстрирующая оба недостатка, приведена на рисунке ниже:



В частности, на рисунке показано, что если функция для вычисления очередного шага моделирования вызывается каждые 20 мс (5 клеточек), а сам процесс моделирования занимает всего 12 мс (3 клетки), то остальные 8 мс процессор простаивает, что является крайне неэффективным.

Кроме того, если после моделирования третьего шага операционной системе потребуется выполнить какой-либо системный процесс, то начало выполнения четвертого шага моделирования будет отложено на неопределенное время. Если в самой процедуре моделирования не проверять, сколько фактически прошло времени, а считать что между двумя последовательными шагами моделирования всегда проходит 20 мс, то результат моделирования будет некорректным, то есть объект сдвинется на меньшее расстояние, чем должен. В результате многократного повторения этой ситуации со временем накопится достаточно большая ошибка, чтобы визуально стало заметно, что объект движется рывками, с переменной скоростью.

Для того чтобы избежать обоих недостатков, описанных выше, применяется другой подход – функция simulation вызывается максимально часто, как только процессор закончил выполнять все предыдущие функции и системные задачи. В этом случае заранее неизвестно, сколько пройдет времени между двумя последовательными вызовами функции simulation, но можно рассчитывать, что это будет минимально возможное время.

Для того чтобы реализовать данный подход с использованием библиотеки glut, требуется зарагестрировать функцию simulation в качестве функции обратного вызова с помощью glutIdleFunc. В этом случае библиотека glut будет вызывать функцию simulation каждый раз после того, как закончит обработку всех сообщений от операционной системы. Обратите внимание, что функция обратного вызова simulation для glutIdleFunc не имеет параметров, в отличие от функции, которая раньше регистрировалась с помощью функции glutTimerFunc.

Этот подход хоть и является оптимальным, ставит новую задачу – необходимость определения в самой функции simulation времени, прошедшего с момента предыдущего шага моделирования (то есть с момента последнего вызова этой функции). Ниже приведена упрощенная структура функции simulation из третьей лабораторной работы:

```
// функция вызывается когда процессор простаивает, т.е. максимально часто
void Simulation(void)
{
	// ОПРЕДЕЛЕНИЕ ВРЕМЕНИ ПРОШЕДШЕГО С МОМЕНТА ПОСЛЕДНЕЙ СИМУЛЯЦИИ В СЕКУНДАХ
	Simulation_Time_Passed = ...

	// ПЕРЕМЕЩЕНИЕ КАМЕРЫ
	bool  CameraLeft        = GetAsyncKeyState(VK_LEFT);
	bool  CameraRight       = GetAsyncKeyState(VK_RIGHT);
	bool  CameraUp          = GetAsyncKeyState(VK_UP);
	bool  CameraDown        = GetAsyncKeyState(VK_DOWN);
	bool  CameraForward     = GetAsyncKeyState(VK_ADD);
	bool  CameraBackward    = GetAsyncKeyState(VK_SUBTRACT);
	camera.setKey(    CameraLeft, CameraRight,
	                  CameraUp, CameraDown,
	                  CameraForward, CameraBackward );
	camera.simulate( Simulation_Time_Passed );

	// ПЕРЕРИСОВАТЬ ОКНО
	glutPostRedisplay();
};
```

Определение времени моделирования.

Для определения времени прошедшего с момента последнего шага в данной работе требуется использовать две функции WinAPI - QueryPerformanceCounter и QueryPerformanceFrequency.

Функция QueryPerformanceCounter возвращает текущее значение счетчика производительности реализуемого операционной системой. Счетчик постоянно увеличивается ядром операционной системы с некоторой частотой, зависящей от производительности компьютера. Таким образом, если запомнить значение этого счетчика на предыдущем шаге (OldValue) и считать значение на новом шаге (NewValue), то разница (NewValue – OldValue) покажет сколько прошло времени между двумя последовательными шагами моделирования. Следует отметит, что прошедшее время выраженно в тиках счетчика производительности, а не в привычных секеундах или милисекундах. Для перевода полученного значения в осмысленные величины необходимо использовать функцию, описанную ниже.

Функция QueryPerformanceFrequency показывает с какой частотой меняется внутренний счетчик производительности, то есть на сколько увеличивается внутренний счетчик производительности за одну секунду. Таким образом, поделив полученную разницу (NewValue – OldValue) на значение частоты, мы получим время между двумя последовательными шагами моделирования в секундах. Полученное значение может применяться для расчета множества параметров, в частности, для расчета того, насколько передвинулся тот или иной объект, если известна его скорость.

Подробнее об этих функциях можно прочитать на сайте Microsoft:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx
https://msdn.microsoft.com/en-us/library/windows/desktop/ms644905(v=vs.85).aspx

Определение состояния клавиш.

Вторая задача, которую необходимо решить в рамках данной лабораторной работы – это определение клавиш, нажатых пользователем. Здесь так же возможности, предоставляемые библиотекой freeglut, являются неудовлетворительными, поэтому необходимо использовать еще одну функцию WinAPI – GetAsyncKeyState. Данная функция имеет множество преимуществ по сравнению с реализацией обработки нажатия клавиши средствами freeglut через механизм обратного вызова. К таким преимуществам, в частности, относятся:

1. Функция позволяет не просто определить была ли нажата клавиша, но и узнать ее текущее состояние – нажата или опущена в текущий момент. Таким образом, существует возможность обработать удержание клавиши.

2. Возможность получить состояние клавиши независимо от выбранного языка и регистра без дополнительной обработки.

3. Возможность получить состояние любой клавиши, в том числе стрелок передвижения курсора, клавиши ENTER и даже левой и правой кнопок мыши.

4. Возможность обработать комбинацию клавиш, например, ctrl + w.

Более подробно с данной функцией можно ознакомиться самостоятельно на сайте Microsoft по следующим ссылкам:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms646293(v=vs.85).aspx
https://msdn.microsoft.com/en-us/library/windows/desktop/dd375731(v=vs.85).aspx

Следует отметить, что механизм обратного вызова glut для обработки клавиш по-прежнему работает и в некоторых случаях проще использовать именно его, например, если требуется обработать именно одинарное нажатие на клавишу, а не её удержание.

Создание класса для работы с камерой.

В рамках данной лабораторной работы необходимо реализовать класс для работы с направленной камерой. Направленная камера – это камера, которая направлена в определенную точку (называемую точкой наблюдения или target), и все движения которой осуществляются вокруг этой точки. Класс для работы с камерой должен реализовывать как минимум следующий функционал:

1. Прежде всего, необходимо реализовать метод, который устанавливает матрицу наблюдения, соответствующей текущей позиции камеры. Непосредственно генерирование матрицы наблюдения выполняется с помощью функции gluLookAt, так же, как это происходило в предыдущих лабораторных работах.

2. Необходимо реализованы основные движения камеры – вращение влево/вправо в горизонтальной плоскости, движение вверх/вниз и приближение/удаление камеры к точке наблюдения. При движении вверх/вниз необходимо предусмотреть граничные углы, выше и ниже которые движение запрещено. Так же при движении вперед/назад необходимо ввести ограничения на максимальное и минимальное расстояние от цели до камеры.

3. Движение камеры должно происходить с определенной скоростью, заданной в виде константы, значение которой должно быть логичным и легко изменяемым. Например, можно задать угловую скорость поворота в градусах - 90 градусов в секунду.

Ниже приведен необходимый минимум методов класса с их кратким описанием. Необходимые поля или дополнительные методы разрабатываются самостоятельно:

```
// КЛАСС ДЛЯ РАБОТЫ С КАМЕРОЙ
class Camera
{
	// необходимые свойства
	...
public:
	// конструктор по умолчанию
	Camera (void);
	// деструктор - сохранение новых параметров камеры
	~Camera (void);
	// установка признаков нажатых клавиш
	void   setKey (bool left, bool right, bool up, bool down, bool forward, bool backward);
	// движение камеры в ранее выбранном направлении
	// параметр - количество секунд прошедших с момента последнего вызова
	void   simulate(float sec);
	// функция для установки матрицы камеры
	void   apply();
};
```

Рассмотрим основные методы, их назначение и логику работы класса:

1. Конструктор (Camera). Конструктор вызывается автоматически при создании нового объекта данного типа. Конструктор необходим для инициализации полей объекта перед его первым использованием, поэтому в конструкторе необходимо установить такие начальные параметры камеры, чтобы на экране были видны все объекты.

2. Метод setKey. В данный метод в качестве параметров передаются признаки того, что пользователь хочет переместить камеру в выбранном направлении. Метод не должен самостоятельно определять нажатие клавиш, поскольку взаимодействие с системой ввода/вывода не является обязанностью класса Camera. Кроме того, если потребуется реализовать передвижение камеры с помощью мышки или джойстика, не надо будет переписывать данный класс. Так же метод не должен непосредственно выполнять передвижение камеры, поскольку он не обладает информацией о прошедшем времени и, следовательно, не знает на какое расстояние необходимо переместить камеру. Метод должен всего лишь запомнить данные признаки во внутренних переменных для последующего использования.
3. Метод simulate является основным методом для передвижения камеры. Метод вызывается в каждом шаге моделирования, где ему в качестве параметра передается время, прошедшее с момента последнего шага симуляции. Изменение параметров камеры происходят именно в этом методе с использованием ранее запомненных признаков необходимости движения в различных направлениях. Отделение метода simulate от метода setKey позволяет реализовать множество дополнительных эффектов, таких как, например, инерционную камеру.
4. Последний метод (apply) устанавливает матрицу камеры соответствующую данной позиции и направлению взгляда. Метод вызывается в функции Display при отрисовке каждого кадра, как это было показано выше. Непосредственно для установки матрицы камеры используется функция gluLookAt. После установки матрицы камеры каждый объект будет домножать на эту матрицу свою собственную матрицу модели для перевода выводимой модели из локальной системы координат в систему координат наблюдателя.

Для защиты лабораторной работы может потребоваться внести одно из изменений перечисленных ниже, поэтому необходимо заранее продумать возможное решение данных заданий и разработать структуру класса так, чтобы эти изменения было легко внести. Сами задания без особых указаний реализовывать не надо. К таким изменениям могут относиться:

1. Реализация инерционной камеры – скорость вращения камеры плавно нарастает (до некоторого предела) после того, как пользователь зажмет клавишу вращения и так же плавно падает до нуля после того, как пользователь отпустит клавишу.
2. Реализация автоматической камеры – после нажатия на клавишу вращения камера начинает вращаться в этом направлении, даже если пользователь отпустил клавишу, до тех пор, пока не будет нажата клавиша вращения в противоположенную сторону.
3. Режим демонстрации – при длительном простое (пользователь не нажимает клавиши) камера начинает самостоятельно вращаться вокруг точки наблюдения.
4. Ускоренное вращение камеры – при зажатой клавише SHIFT вращение камеры происходит в два раза быстрее.
5. Реализация нескольких камер. В системе есть несколько камер, переключение между которыми выполняется по нажатию на клавишу TAB.
6. Некоторые дополнительные задания.

Задание к лабораторной работе.

Лабораторная работа №3 строится на основе предыдущей работы с внесением следующих изменений:

1. В отдельном модуле реализовать класс Camera вышеописанной структуры для работы с направленной камерой. Камера должна выполнять требования, описанные в соответствующем разделе. Для лучшего понимания принципов передвижения камеры можно обратиться к демонстрационному примеру.
2. Привести функцию Display к виду описанному выше. При необходимости в функцию можно добавить дополнительные возможности, однако общий принцип должен оставаться неизменным – функция Display служит только для вывода изображения на экран и не должна модифицировать никакие глобальные данные.
3. Реализовать функцию Simulation. Функция должна регистрироваться с помощью glutIdleFunc, определять состояние требуемых клавиш, время симуляции и вызывать соответствующие методы для изменения позиции камеры.

Содержание отчета.

1. Титульный лист.
2. Задание к лабораторной работе.
3. Текст h и cpp файлов модуля класса с классом CCamera.
4. Текст основной программы с комментариями.
5. Скриншот работы программы.

Критерии оценки и вопросы к защите.

Вопросы по теоретической части (повторение):

1.1 Что такое модель, объект, сцена?
1.2 Что такое локальная система координат (система координат модели)?
1.3 Что такое глобальная система координат (мировая система координат)?
1.4 Что такое система координат наблюдателя (видовая система координат)?
1.5 Для чего применяется матрица модели, матрица наблюдения и матрица проекции?
1.6 Напишите последовательность матричных операций по преобразованию вершины модели для вывода её на экран?

Вопросы по практической части:

2.1 Опишите для чего нужна команда gluLookAt и назначение её параметров.
2.2 Опишите для чего нужна команда gluPerspective и назначение её параметров.
2.3 Опишите принцип работы команды glLoadIdentity и объясните, зачем она используется при установке матрицы проекции и матрицы камеры.

Требования к программе:

3.1 Реализация класса Camera в отдельном модуле;
3.2 В программе должны отсутствовать «магические» числа. Все параметры алгоритмов передвижения камеры должны быть объявлены как константы. Эти константы должны носить осмысленные имена и значения. Например, скорость вращения камеры должна задаваться в градусах в секунду, а не в некоторых абстрактных единицах измерения.
3.3 К каждому методу класса должен быть комментарий;
3.4 Глобальные переменные должны носить осмысленные имена;

GOOD CODERS...