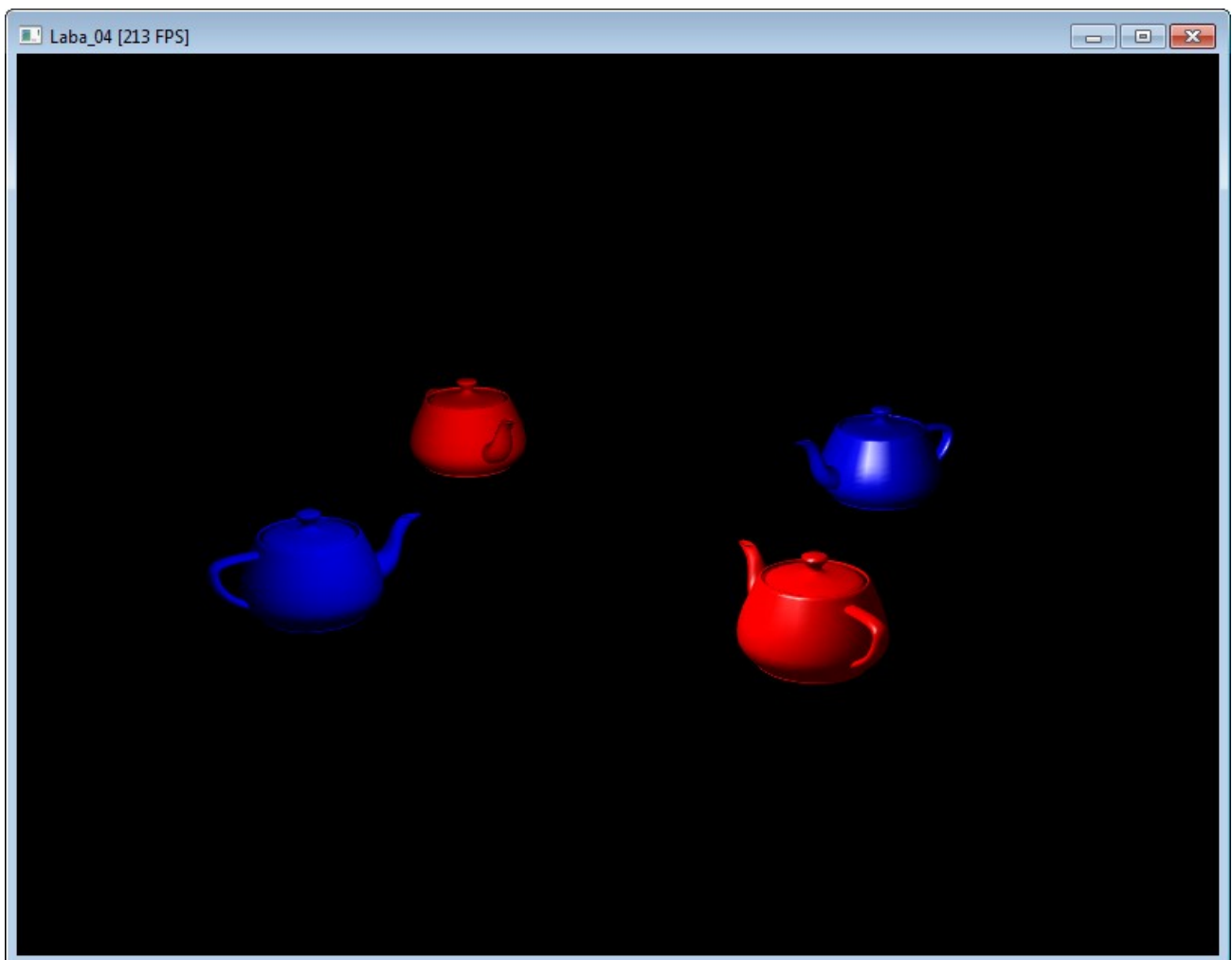


# Лабораторная работа №5

## Реализация освещения по методу Фонга.

В пятой лабораторной работе необходимо реализовать освещение сцены по методу Фонга. Для расчета освещения необходимо установить один источник света, который является общим для всех объектов, а также установить параметры материала для каждого объекта. Параметры материала для каждого объекта должны задаваться в отдельном текстовом файле, который необходимо загрузить на этапе инициализации программы. Использование внешнего файла позволит, с одной стороны, настраивать внешний вид графических объектов (их цвет), без необходимости перекомпилировать программу, а, с другой стороны, позволит уменьшить код по инициализации данных.

Для реализации освещения в лабораторной работе необходимо создать два класса: класс для представления источника света (Light) и класс для представления материала (Material). Особое внимание необходимо уделить способу организации взаимосвязи между графическим объектом и материалом. В частности, необходимо сделать так, чтобы каждый графический объект обладал каким-либо материалом, и при этом несколько графических объектов могли использовать один и тот же материал.



## Цели лабораторной работы и порядок её выполнения.

Лабораторная работа №5 строится на основе предыдущей работы с внесением необходимых дополнений. При этом, к основным целям лабораторной работы относятся:

1. **Реализация освещения по методу Фонга.** Необходимо изучить функции OpenGL для реализации освещения с использованием модели освещения Фонга, иметь представление о различных компонентах освещения в соответствии с моделью Фонга, а также знать, какое влияние оказывает каждый компонент на итоговый цвет объектов.
2. **Реализация классов и работа с файлами.** Создание классов на этом этапе уже не должно представлять сложность, поэтому следующей целью является инициализация объектов класса, путем считывания параметров этих объектов из внешних файлов. В частности, в лабораторной работе необходимо сделать так, чтобы параметры материала для каждого графического объекта задавались не жестко в тексте программы, а хранились во внешних файлах, что делает настройку программы (подбор цвета объектов) тривиальной задачей, не требующей повторной компиляции программы.
3. **Ознакомление со способами организации взаимосвязи между объектами.** В сложных программах каждый класс должен реализовывать свою часть функциональности, что позволит реализовывать или модифицировать каждый класс независимо от других классов (single responsibility principle). Тем не менее, некоторые классы для своего функционирования полагаются на другие классы. Например, в рамках данной лабораторной работы необходимо сделать так, чтобы при выводе каждого графического объекта устанавливался его материал.

Для реализации зависимости одного класса от другого используются такие понятия как: композиция, агрегация и делегирование. Данные понятия используются при программировании в любой сфере и с использованием любого языка программирования, поэтому получение практических навыков в реализации этих принципов – это, пожалуй, главная цель данной лабораторной работы!

4. **Получение практических навыков работы с умными указателями (smart pointers).** Умные указатели в C++ позволяют сделать работу с динамической памятью более безопасной, за счет отказа от самостоятельного выделения и удаления динамической памяти. В настоящее время использование умных указателей считается хорошим тоном, говорящим о достаточно высокой квалификации C++ программиста (необходимое, но увы не достаточное условие).

При изучении темы умных указателей необходимо разобраться с тем, какие проблемы они призваны решить и для чего предназначен каждый вид указателей. После этого можно переходить к изучению синтаксиса создания указателей и практики работы с ними.

Перед выполнением лабораторной работы необходимо полностью прочитать методические указания, включая задание к лабораторной работе. В случае, если после прочтения методических указаний лабораторная работа кажется достаточно легкой, её можно выполнять в той последовательности, которая кажется более удобной. В противном случае лабораторную работу рекомендуется выполнять в соответствии со следующей последовательностью действий:

1. **Предварительная настройка.** Для отладки программы рекомендуется создать демонстрационную сцену, содержащую один чайник, расположенный в центре сцены. При выводе чайника необходимо вместо проволочного чайника (glutWireTeapot) выводить сплошной чайник (glutSolidTeapot).

Далее необходимо включить механизм расчета освещения. Поскольку еще не включен ни один источник освещения, чайник должен быть тёмно-серым, без ярко выраженных граней объекта. Для того, чтобы лучше различать чайник, можно поменять цвет фона на более яркий.

2. **Реализовать класс для работы с источником света.** Далее необходимо реализовать класс для работы с источником света (Light), создать глобальный объект этого типа, инициализировать созданный объект и в функции display вызвать метод для передачи параметров источника света в OpenGL (метод apply). Поскольку в OpenGL существует материал по умолчанию, то (если все сделано корректно) чайник должен казаться освещенным с плавным изменением затененности.

При задании параметров источника света особое внимание необходимо обратить на установку его позиции. Прежде всего, позицию источника света необходимо задать так, чтобы он располагался над горизонтальной плоскостью на некотором удалении от чайника, чтобы чайник освещался сверху и немного сбоку. В демонстрационном примере источник света установлен в позиции (20.0, 20.0, 15.0).

Также следует отметить, что при передаче позиции источника света в OpenGL переданный вектор будет автоматически умножен на текущую матрицу модели, поэтому параметры источника света необходимо устанавливать сразу после установки камеры, но до вывода графических объектов.

3. **Реализовать класс для работы с материалом.** Далее необходимо реализовать класс для представления материала и создать одну глобальную переменную данного типа. На начальном этапе можно не реализовывать загрузку материала из файла, а жестко установить значение полей объекта на этапе инициализации программы.

Если перед выводом графического объекта применить созданный материал, то объект должен выводиться с соответствующими параметрами. Для проверки можно попробовать задать желтый или фиолетовый материал, чтобы убедиться, что результирующее изображение совпадает с ожидаемым.

4. **Реализовать загрузку параметров материала из внешнего файла.** Далее необходимо реализовать функцию загрузки параметров материала из файла. Пример файла, задающего параметры материала, представлен в демонстрационной программе.

Для загрузки данных из файла рекомендуется воспользоваться классом `ifstream`. При необходимости рекомендуется почитать дополнительную литературу или посмотреть видео-уроки. В процессе загрузки нелишним будет выводить считанные данные на консоль, чтобы убедиться в правильности процесса загрузки.

5. **Организовать взаимосвязь между графическим объектом и материалом.** Затем необходимо организовать взаимодействие между графическим объектом и материалом. В частности, необходимо сделать так, чтобы графический материал имел указатель на используемый материал и при выводе графического объекта на экран вызывался соответствующий метод материала для передачи его параметров в OpenGL. Это позволит сделать так, чтобы для каждого графического объекта можно было назначить свой материал.

Для проверки рекомендуется создать несколько различных материалов и посмотреть, меняется ли результат работы программы, если для графического объекта устанавливать различные материалы.

6. **Разобраться с умными указателями.** Далее необходимо разобраться с умными указателями. В процессе изучения материала рекомендуется написать небольшую тестовую программу, чтобы убедиться в правильности понимания того, как они работают. Умные указатели будут использоваться во всех последующих работах, поэтому необходимо добиться уверенного владения данным механизмом C++.
7. **Реализовать использование умных указателей в программе.** Затем необходимо внести изменения в класс `GraphicObject` так, чтобы для указания на используемый материал применялись умные указатели, в частности `shared_ptr`, а не обычные указатели C++ (raw pointers).
8. **Полностью реализовать задание к лабораторной работе.** В завершении необходимо полностью реализовать задание к лабораторной работе, в частности, вывести четыре чайника с разными материалами. Для этого требуется загрузить все четыре материала из демонстрационного примера и назначить их нужным графическим объектам. Если все реализовано корректно, то результат работы программы должен совпадать с демонстрационным параметром.

## Модель освещения по методу Фонга.

Модель освещения Фонга была предложена в попытке относительно простыми формулами добиться возможности представлять достаточно большой спектр материалов. Речь, прежде всего, идет о возможности задавать как матовые объекты, освещенность которых не зависит от взаимного расположения наблюдателя и источника света, так и глянцевые, которые отражают большинство лучей в определенном направлении, формируя блик на поверхности:



Модель Фонга не является фотореалистичной, то есть формулы, используемые для расчета, являются весьма далекими от физически обоснованных. Тем не менее, к преимуществам модели Фонга можно отнести то, что она является весьма простой, что позволяет быстро рассчитывать освещенность каждой вершины параллельно и независимо друг от друга.

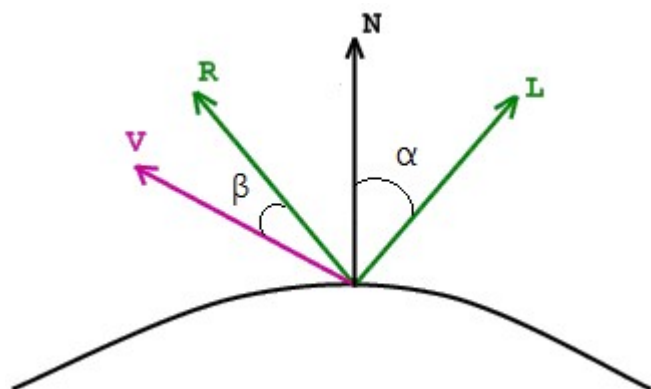
Для представления различных аспектов материала свет делят на три компонента:

- Рассеянный компонент света (диффузный, diffuse).
- Зеркальный компонент света (спекулярный, specular).
- Фоновый компонент света (амбиентный, ambient).

Следует отметить, что каждый компонент света кодируется с помощью четверки ( $r, g, b, a$ ). Таким образом можно говорить, что, например, источник света испускает ярко красные диффузные лучи (диффузная составляющая источника света равна  $(1.0, 0.0, 0.0, 1.0)$ ), белые specularные лучи (specularная составляющая равна  $(1.0, 1.0, 1.0, 1.0)$ ) и имеет незначительное количество темно-красных ambientных лучей (ambientная составляющая равна  $(0.4, 0.0, 0.0, 1.0)$ ). Последний компонент в четверке чисел называется альфа-компонентой, определяет, обычно, степень прозрачности и здесь не используется.

Материал объекта также состоит из трех компонент, которые задают коэффициент отражения для рассеянных, specularных и фоновых лучей источника света соответственно. Это позволяет, например, указать, что материал не отражает зеркальные лучи (specularная составляющая равна  $(0.0, 0.0, 0.0, 1.0)$ ). В этом случае объект будет освещен одинаково, независимо от направления взгляда и будет казаться матовым. Или, наоборот, указать, что объект отражает почти все зеркальные лучи (specularная составляющая равна  $(0.9, 0.9, 0.9, 1.0)$ ), что приведет к тому, что объект будет восприниматься глянцевым.

По методу Фонга освещенность (цвет) для любой точки вычисляется в соответствии с рисунком, представленным ниже:



L – вектор на источник света;

R – отраженный относительно нормали вектор L (вектор идеального отражения);

V – направление на наблюдателя;

N – вектор нормали к поверхности (задает ориентацию поверхности в данной точке);

Вклад каждой составляющей освещенности, а также формула её расчета представлена далее.

### Рассеянная (диффузная, diffuse) составляющая.

Диффузные лучи — это лучи, которые при падении на поверхность отражаются равномерно во все стороны. Освещенность объекта при этом зависит от угла падения луча света на поверхность, но не зависит от положения наблюдателя. Итоговая формула освещенности для лучей данного типа представлена ниже:

$$I_d = L_d * M_d * \cos(\alpha)$$

$\alpha$  – угол падения луча света относительно нормали;

$L_d$  – диффузная составляющая источника света;

$M_d$  – диффузная отражающая способность материала;

Освещенность данного типа не зависит от позиции наблюдателя и является основной составляющей освещенности. Если говорится про красный объект, то, прежде всего, имеется в виду, что данный объект отражает преимущественно красные диффузные лучи, то есть диффузная составляющая материала будет иметь значение близкое к (1.0, 0.0, 0.0, 1.0). Остальные компоненты, например, зеркальная, могут быть или их может не быть. В этом случае объект будет считаться либо красным матовым, либо красным глянцевым.

### Зеркальная (спекулярная, specular) составляющая.

Спекулярные лучи – это лучи, которые ведут себя очень похоже на идеальные, то есть отражаются не во все стороны, а вдоль вектора идеального отражения, поэтому они влияют на воспринимаемую освещенность объекта только в том случае, если наблюдатель расположен в указанном направлении. Данная составляющая формирует блик на поверхности. Итоговая формула зеркальной составляющей представлена ниже:

$$I_s = L_s * M_s * \cos^p(\beta)$$

$\beta$  – угол между идеально отраженным лучом и вектором на наблюдателя;

$L_s$  – зеркальная составляющая источника света;

$M_s$  – зеркальная отражающая способность материала;

$p$  – степень отполированности материала (сконцентрированность блика).

Как правило, способностью отражать зеркальную составляющую, обладают глянцевые объекты, такие как стекло или неокрашенный металл. Объекты подобного рода имеют ярко выраженную зеркальную составляющую. Матовые объекты, как правило, её не имеют, то есть их зеркальная составляющая близка к (0.0, 0.0, 0.0, 1.0). Зеркальная составляющая часто определяется как некоторая часть (процент) от диффузной составляющей, чтобы основной цвет объекта и цвет блика были одинаковыми.

Зеркальная составляющая – это единственная составляющая, чей вклад зависит от положения наблюдателя, в частности, от угла  $\beta$ . Кроме того, вычисленный коэффициент освещенности ( $\cos(\beta)$ ) возводится в некоторую степень, поэтому зеркальная составляющая отвечает только за блик на поверхности объекта и быстро уменьшается до нуля при удалении наблюдателя в сторону от вектора R.

#### Фоновая (амбиентная, ambient) составляющая.

Амбиентные лучи – это вторичные лучи, то есть лучи, которые падают на объект со всех сторон, после того как были отражены другими объектами. Фоновая составляющая необходима для того, чтобы поверхности, на которые не падают прямые лучи от источника света, все равно не были абсолютно черными, а были хотя бы чуть-чуть освещены. Интенсивность таких лучей, как правило, невелика. Например, для красного объекта она может быть равна (0.2, 0.0, 0.0, 1.0). Формула расчета выглядит следующим образом:

$$I_a = L_a * M_a$$

$L_a$  – фоновая составляющая источника света;

$M_a$  – фоновая отражающая способность материала;

#### Самосвечение (emission).

Иногда к модели освещения Фонга добавляют самосвечение материала, то есть способность материала не просто отражать падающий на него свет, но и самостоятельно испускать лучи света. Это справедливо, например, для фар автомобиля, неоновых вывесок или экранов компьютеров. В этом случае цвет самосвечения добавляется к итоговому значению. Сама формула для расчета компонента, отвечающего за самосвечение, приведена ниже:

$$I_e = M_e$$

$M_e$  – параметр материала, задающий цвет испускаемых лучей.

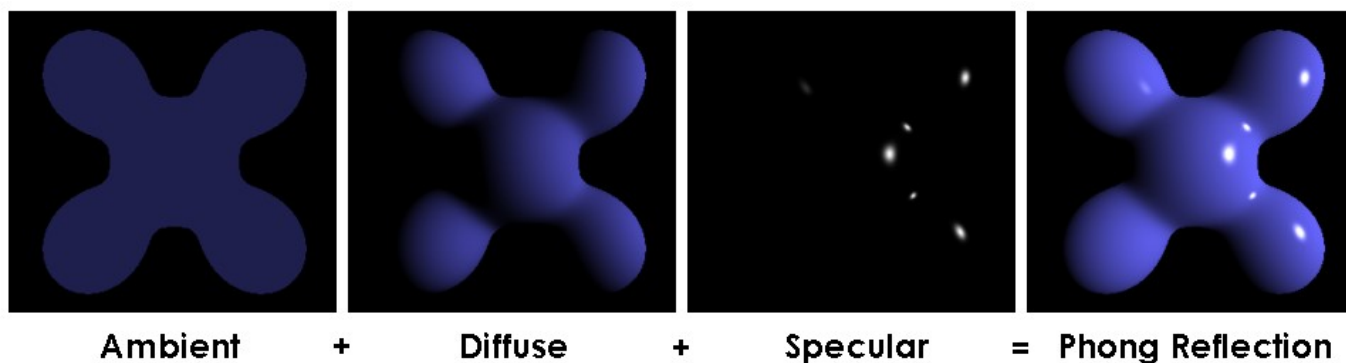
#### Итоговая формула.

Итоговая формула освещенности для одного источника света выглядит следующим образом:

$$I = L_d * M_d * \cos(\alpha) + L_s * M_s * \cos^p(\beta) + L_a * M_a + M_e$$

Библиотека OpenGL поддерживает возможность рассчитывать освещенность сцены сразу от нескольких источников света (любая реализация OpenGL должна поддерживать не менее восьми источников света). В этом случае вычисленные по формуле выше интенсивности освещения складываются для каждого источника света. Чем больше источников света – тем ярче освещены объекты, что, в целом, является весьма логичным.

Влияние каждой компоненты модели освещения Фонга (кроме самосвечения) показано ниже:



### Влияние вектора нормали.

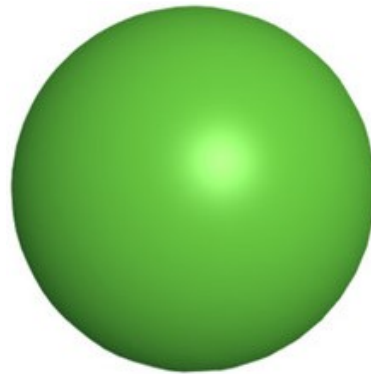
Отдельного внимания заслуживает вектор нормали. Несмотря на то, что в данной лабораторной работе не требуется самостоятельно вычислять или передавать нормаль при выводе графического объекта (функция `glutSolidTeapot` делает это автоматически), необходимо иметь представление о влиянии нормали на итоговое освещение.

Существует два способа задания векторов нормалей.

Первый способ – это истинные нормали, то есть это такие вектора нормалей, которые действительно являются перпендикулярными к полигону, в который входит данная вершина. В этом случае заметно изменение освещенности на границе двух полигонов. Такой способ, который иногда называют плоским затенением, может применяться для обозначения объектов, которые имеют угловатую форму:



Второй способ – это использование усредненных нормалей. В этом случае вычисляется усредненный вектор нормали для всех соседних полигонов. В результате резкое изменение освещенности не происходит, хотя по контуру видно, что объект действительно является угловатым и состоит из множества плоских полигонов:



В современной практике используются оба способа в зависимости от того, какого эффекта необходимо добиться. Второй способ требует больше предварительной работы для определения усредненных нормалей и может применяться далеко не ко всем объектам.

## Реализация освещения по методу Фонга средствами OpenGL.

В OpenGL для вывода объектов с использованием модели освещения Фонга необходимо настроить конвейер рендеринга, выполнив последовательность шагов, представленных ниже. При этом следует иметь в виду, что для корректного результата правильно должен быть выполнен каждый этап:

- Включить модель освещения Фонга и настроить общие параметры модели расчета освещенности при необходимости (если значение по умолчанию не удовлетворяет требованиям программы);
- Включить источник света и задать его параметры. Особое внимание следует уделить установке позиции источника света, поскольку OpenGL ожидает, что все координаты, в конечном счете, будут выражены в общей системе координат, в качестве которой используется система координат наблюдателя (view space);
- Задать параметры материала. Перед выводом каждого объекта можно устанавливать разные параметры материала. Таким образом, каждый графический объект может иметь свой собственный материал и, соответственно, на экране выглядеть по-разному.
- Вывести объект, передавая в OpenGL вектор нормали для каждой вершины. Для расчета освещения необходимо, чтобы вершина, наряду с прочими атрибутами, имела также и вектор нормали, который будет использоваться OpenGL для расчета косинуса углов  $\alpha$  и  $\beta$ . В рамках данной лабораторной работы для вывода модели будет использоваться функция `glutSolidTeapot`, которая самостоятельно передает в OpenGL вектора нормалей для каждой вершины.

### Включение и настройка модели освещения Фонга.

В OpenGL существует множество механизмов, большая часть которых по умолчанию отключена. Одним из таких механизмов является механизм расчета освещенности вершины по методу Фонга. По умолчанию эта возможность отключена и в качестве цвета вершины используется цвет, заданный напрямую, например, через функцию `glColor*`. Включение любых механизмов осуществляется с помощью функции `glEnable`, параметр которой указывает, какой именно механизм требуется включить.

Для включения механизма расчета освещения по методу Фонга используется следующая команда:

```
// включаем режим расчета освещения
glEnable(GL_LIGHTING);
```

Следует отметить, что расчет освещенности требует множества параметров, например, параметров материала и источника света, а также указания вектора нормали для каждой вершины. Если любой из этих параметров установлен некорректно, то все объекты, скорее всего, будут выведены абсолютно черными.

В случае если в каком-то механизме больше нет необходимости, он может быть отключен, используя функцию `glDisable`, с указанием в качестве параметра той же самой константы. Совершенно нормальной является ситуация, когда часть объектов выводятся с использованием модели Фонга, а часть объектов (например, пользовательский интерфейс) - без использования данной модели.

Помимо параметров материала и источника света, OpenGL позволяет задать ряд параметров, относящихся к самой модели освещения. Данные параметры влияют на все объекты сцены. Все параметры модели освещения задаются с помощью одной и той же команды `glLightModel*`, в которой указывается имя задаваемого параметра (одна из предопределенных констант) и само значение параметра. Для задания всех параметров модели освещения требуется вызвать данную функцию несколько раз. Для примера ниже приводится код, задающий общую фоновую освещенность:

```
// устанавливаем общую фоновую освещенность
GLfloat globalAmbientColor[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globalAmbientColor);
```



Прототип функции выглядит следующим образом:

```
void glLightModelfv (GLenum pname, const GLfloat * params);
```

- Первый параметр (pname) – «имя» задаваемого параметра, которое указывает, какой аспект модели освещения задается в данный момент. «Имя» параметра указывается с помощью константы;
- Второй параметр (params) - указатель на массив, содержащий данные, которые будут установлены в качестве значения указанного параметра.

Следует отметить, что все параметры модели освещения имеют значения по умолчанию, которые, в большинстве случаев, являются приемлемыми для любой программы. Более того, некоторые параметры влияют только на редко используемые аспекты и в данной работе использоваться не будут, поэтому данную функцию можно не вызывать.

Краткое описание параметров представлено в следующей таблице:

«Имя» параметра	Назначение параметра
GL_LIGHT_MODEL_AMBIENT	Вектор (r, g, b, a) задающий общую фоновую освещенность всех объектов. Значение по умолчанию: (0.2, 0.2, 0.2, 1.0), поэтому даже при выключенном источнике света объекты все равно немного освещены.
GL_LIGHT_MODEL_COLOR_CONTROL	Данный параметр определяет, как будет рассчитываться итоговый цвет. Параметр может принимать следующие значения из списка: <ul style="list-style-type: none"><li>• GL_SINGLE_COLOR – рассчитывается общий цвет;</li><li>• GL_SEPARATE_SPECULAR_COLOR - зеркальный цвет рассчитывается отдельно и прибавляется после наложения текстуры.</li></ul> Параметр не оказывает влияния, если текстурирование не используется. При использовании текстурирования лучше использовать второе значение параметра (GL_SEPARATE_SPECULAR_COLOR).
GL_LIGHT_MODEL_LOCAL_VIEWER	Параметр определяет способ расчета зеркальной составляющей и может принимать два значения: <ul style="list-style-type: none"><li>• 0 – используется упрощенная модель расчета отраженного луча;</li><li>• Любое значение отличное от нуля – используется более точное вычисление отраженного луча.</li></ul>
GL_LIGHT_MODEL_TWO_SIDE	Определяет, используется ли двухсторонний материал. Данный параметр может принимать два значения: <ul style="list-style-type: none"><li>• 0 – используется только материал для лицевых граней;</li><li>• Любое значение отличное от нуля – используется как материал для лицевых граней, так и материал для нелицевых граней. Кроме того, для нелицевых граней автоматически инвертируется вектор нормали.</li></ul> Данный параметр оказывает влияние только в том случае, если нелицевые грани видны наблюдателю, например, при выводе полупрозрачных объектов. Значение по умолчанию – 0.

Более подробную информацию можно получить по следующей ссылке:

<https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glLightModel.xml>

## Установка параметров источника света.

OpenGL позволяет рассчитывать освещенность сразу от нескольких источников света (не менее восьми), поэтому необходимо указать, какие именно источники света следует принимать во внимания. Иначе говоря, необходимо «включить» определенные источники света. Для этого применяется функция `glEnable`, параметром которой в этот раз является номер источника света, заданный специальной константой. Например, для включения нулевого источника света используется следующая команда:

```
// включаем нулевой источник света
glEnable(GL_LIGHT0);
```

Далее необходимо задать параметры источника света. Все параметры источника света задаются с помощью функции `glLight*`, один из вариантов которой приведен ниже:

```
void glLightfv (GLenum light, GLenum pname, const GLfloat *params);
```

- Первый параметр (`light`) – определяет номер источника света с помощью одной из констант вида `GL_LIGHT0`, `GL_LIGHT1` и так далее;
- Второй параметр (`pname`) – «имя» задаваемого параметра, которое указывает, какой аспект источника света задается в данный момент. «Имя» параметра указывается с помощью констант;
- Третий параметр (`params`) - указатель на массив, содержащий данные, которые будут установлены в качестве значения указанного параметра.

Источник света обладает множеством параметров, каждый из которых имеет некоторое значение по умолчанию. Для установки всех необходимых параметров требуется вызвать функцию `glLight*` несколько раз. Краткое описание параметров представлено в следующей таблице:

«Имя» параметра	Назначение параметра
GL_AMBIENT	Фоновая составляющая освещенности. Вектор из четырех вещественных компонент (r, g, b, a).
GL_DIFFUSE	Диффузная составляющая освещенности. Вектор из четырех вещественных компонент (r, g, b, a).
GL_SPECULAR	Зеркальная составляющая освещенности. Вектор из четырех вещественных компонент (r, g, b, a).
GL_POSITION	Позиция источника света. Вектор из четырех вещественных компонент (x, y, z, w).
GL_SPOT_DIRECTION GL_SPOT_EXPONENT GL_SPOT_CUTOFF	Данные параметры позволяют определить прожекторный эффект, то есть эффект, при котором свет от источника света распространяется не во все стороны, а только в определенном направлении. Значения по умолчанию подобраны таким образом, чтобы данный эффект себя не проявлял. В данной лабораторной работе эти параметры не используются.
GL_CONSTANT_ATTENUATION GL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	Данные параметры позволяют определить эффект затухание света с расстоянием, то есть чем дальше объект находится от источника света, тем менее он освещен. Значения по умолчанию подобраны таким образом, чтобы данный эффект не проявлялся. В данной лабораторной работе эти параметры не используются.

Более подробную информацию можно получить по следующей ссылке:

<https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glLight.xml>

Отдельно следует упомянуть позицию источника света. Позиция задается вектором из 4-х компонент вещественного типа – (x, y, z, w). При этом последняя компонента w, как правило, равна либо единице, либо нулю. В первом случае источник света считается точечным и его координаты задаются компонентами (x, y, z). Во втором либо координата w равна нулю, в этом случае вектор (x, y, z) задаёт направление на источник света. Последний случай можно использовать для задания направленных источников света, например, таких как солнце.



Следует иметь в виду, что позиция источника света, при передаче её в OpenGL, автоматически умножается на текущую матрицу наблюдения модели. Это сделано для того, чтобы можно было легко перевести координаты источника света из глобальной системы координат (в которой они обычно хранятся) в систему координат наблюдателя. Поэтому, прежде чем передавать позицию источника света в OpenGL, необходимо установить матрицу камеры. Только после этого можно задавать позицию источника света, что приведет к тому, что позиция будет умножена на текущую матрицу наблюдения модели (в данном случае матрицу камеры) и сохранена внутри OpenGL в системе координат наблюдателя.

Также следует отметить, что для лучшего результата источник света необходимо разместить так, чтобы он освещал все объекты, то есть его лучше расположить над плоскостью на некоторой высоте, желательно сдвинутым относительно центра плоскости, чтобы грани были освещены неравномерно.

Для удобства работы с источником света необходимо реализовать отдельный класс со следующей структурой:

```
// КЛАСС ДЛЯ РАБОТЫ С ИСТОЧНИКОМ СВЕТА
class Light
{
public:
    // конструкторы
    Light();
    Light(vec3 position);
    Light(float x, float y, float z);

    // задание различных параметров источника света
    void setPosition (vec3 position);
    void setAmbient (vec4 color);
    void setDiffuse (vec4 color);
    void setSpecular (vec4 color);

    // установка всех параметров источника света с заданным номером
    // данная функция должна вызываться после установки камеры,
    // т.к. здесь устанавливается позиция источника света
    void apply (GLenum LightNumber = GL_LIGHT0);

private:
    // позиция источника света
    vec4 position;
    // фоновая составляющая источника света
    vec4 ambient;
    // диффузная составляющая
    vec4 diffuse;
    // зеркальная составляющая
    vec4 specular;
};
```

Назначение полей и методов очевидно по их названиям и приведенным комментариям. Наиболее важным методом является метод apply, который используется для передачи в OpenGL всех параметров источника света – это то, ради чего и создавался данный класс. При необходимости можно добавить дополнительные методы, например, метод для установки коэффициентов затухания или геттеры для всех полей класса.

## Установка параметров материала.

Для задания параметров материала используется одна из следующих функций, выбираемая в зависимости от передаваемого параметра (одно значение или указатель на массив, содержащий несколько значений):

```
void glMaterialf( GLenum face, GLenum pname, GLfloat param);
```

```
void glMaterialfv( GLenum face, GLenum pname, const GLfloat *params);
```

- Первый параметр (face) - указывает, для каких граней применяется материал. Можно задать два разных материала для лицевых и для нелицевых граней. В качестве значений выступают константы GL\_FRONT, GL\_BACK или GL\_FRONT\_AND\_BACK.
- Второй параметр (pname) - задает «имя» устанавливаемого параметра материала, то есть указывает, какой именно компонент материала устанавливается с помощью этой функции.
- Последний параметр (param) - непосредственно содержит значение, которое будет присвоено указанному параметру.

Имена параметров задаются константой и представлены в таблице, приведенной ниже. Следует отметить, что для того, чтобы установить все параметры материала, данную функцию придется вызывать пять раз:

«Имя» параметра	Назначение параметра
GL_AMBIENT	Фоновая отражающая способность материала. Вектор из 4-х компонент вещественного типа (r, g, b, a). Последний компонент, альфа, для фоновой составляющей, как правило, не используется и обычно равен единице. Значение по умолчанию: (0.2, 0.2, 0.2, 1.0).
GL_DIFFUSE	Диффузная отражающая способность. Вектор из 4-х компонент вещественного типа (r, g, b, a). Последний компонент, альфа, для диффузной составляющей обычно обозначает степень прозрачности и меняется от нуля (объект полностью прозрачен) до единицы (объект полностью непрозрачен). Однако без использования специального режима OpenGL данное значение не оказывает влияние на результирующее изображение. Тем не менее, на данном этапе рекомендуется ставить значение равно 1. Значение по умолчанию: (0.8, 0.8, 0.8, 1.0).
GL_SPECULAR	Зеркальная отражающая способность. Вектор из 4-х компонент вещественного типа (r, g, b, a). Альфа компонент для зеркальной составляющей, как правило, не используется и обычно равен единице. Значение по умолчанию: (0.0, 0.0, 0.0, 1.0).
GL_EMISSION	Цвет самосвечения. Вектор из 4-х компонент вещественного типа (r, g, b, a). Как правило, альфа компонент для самосвечения не используется и обычно равен единице. Значение по умолчанию: (0.0, 0.0, 0.0, 1.0).
GL_SHININESS	Степень отполированности объекта. Коэффициент $p$ в формуле. Вещественное значение. Может принимать значение от 0 до 128, для более выраженного блика рекомендуется использовать значения больше единицы, например, 32 или 64. Значение по умолчанию: 0.

Более подробную информацию можно получить по следующей ссылке:

<https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glMaterial.xml>

Для работы с материалом необходимо организовать класс со следующей структурой:

```
// КЛАСС ДЛЯ РАБОТЫ С МАТЕРИАЛОМ
class PhongMaterial
{
public:
    // конструктор по умолчанию
    PhongMaterial();

    // задание параметров материала
    void setAmbient(vec4 color);
    void setDiffuse(vec4 color);
    void setSpecular(vec4 color);
    void setEmission(vec4 color);
    void setShininess(float p);

    // загрузка параметров материала из внешнего текстового файла
    void load(std::string filename);

    // установка всех параметров материала
    void apply();

private:
    // фоновая составляющая
    vec4 ambient;
    // диффузная составляющая
    vec4 diffuse;
    // зеркальная составляющая
    vec4 specular;
    // самосвечение
    vec4 emission;
    // степень отполированности
    float shininess;
};
```

Отдельного внимания заслуживает метод load, который позволяет загрузить параметры материала из внешнего файла. В частности, для задания параметров материала требуется вызвать пять различных методов данного класса, что может выглядеть следующим образом:

```
// задание всех параметров одного материала
PhongMaterial material;
material.setDiffuse(vec4(1.0, 0.0, 0.0, 1.0));
material.setAmbient(vec4(0.4, 0.0, 0.0, 1.0));
material.setSpecular(vec4(1.0, 1.0, 1.0, 1.0));
material.setEmission(vec4(0.0, 0.0, 0.0, 1.0));
material.setShininess(64.0);
```

С учетом того, что в данной работе требуется установить параметры для четырех материалов, код начинает неоправданно разрастаться. Кроме того, для изменения параметров материала потребуется перекомпилировать исходный код программы, что может быть неудобно при настройке внешнего вида объектов. Поэтому необходимо реализовать метод load, который позволяет загружать все параметры материала из внешнего текстового файла (структура файла приведена в демонстрационном примере).

В этом случае установка параметров материала может быть выполнена гораздо короче:

```
// задание всех параметров одного материала
PhongMaterial material;
material.load("data/materials/material_1.txt");
```

В случае необходимости, дополнительные сведения по работе с файлами можно найти по следующим ссылкам:

[https://www.youtube.com/watch?v=CBnB2fvfu\\_I&list=PLQOaTSbfxUtCrKs0nicOg2npJQYSPGO9r&index=134](https://www.youtube.com/watch?v=CBnB2fvfu_I&list=PLQOaTSbfxUtCrKs0nicOg2npJQYSPGO9r&index=134)  
<https://www.youtube.com/watch?v=aUP0eAEIxog&list=PLQOaTSbfxUtCrKs0nicOg2npJQYSPGO9r&index=135>

## Определение связи между графическим объектом и материалом (v 1.0).

Для облегчения разработки, программы разбиваются на отдельные, относительно независимые классы, каждый из которых решает только одну задачу в соответствии с принципом единственной ответственности. Данный подход облегчает реализацию классов, позволяет работать над проектом нескольким программистам одновременно, а также улучшает тестируемость кода. Тем не менее, для получения конечного результата необходимо организовать взаимодействие классов между собой, чтобы совместными усилиями они решали поставленную задачу.

В объектно-ориентированных языках программирования существует несколько способов организации взаимодействия между классами.

**Наследование** — это способ организации взаимодействия между классами, при котором класс-наследник имеет все поля и методы родительского класса, и, как правило, добавляет какой-то дополнительный функционал. Наследование описывается словом «является» (is). Например, класс «легковой автомобиль» может являться потомком более общего класса «автомобиль». Данный способ взаимодействия классов может применяться если классы, с точки зрения логики, действительно принадлежат одной группе.

**Ассоциация** — это такой способ организации взаимодействия между классами, когда один класс включает в себя другой класс в качестве одного из полей. Ассоциация описывается словом «имеет» (has). Например, класс «автомобиль» одним из своих полей может иметь объект класса «двигатель».

Выделяют два частных случая ассоциации: композицию и агрегацию.

**Композиция** — это способ организации взаимодействия между классами, при котором один объект является неотъемлемой частью другого объекта и не может существовать отдельно от него. Например, можно сказать, что «двигатель» не существует отдельно от «автомобиля». В этом случае «двигатель» создается при создании «автомобиля» и полностью управляется автомобилем.

**Агрегация** — иной способ организации взаимодействия между классами, когда первый объект, хоть и связан со вторым, но, тем не менее, может существовать отдельно. Например, экземпляр «двигателя» создается отдельно, и передается в «автомобиль» в виде параметра. В этом случае в «автомобиле» хранится только указатель (ссылка) на «двигатель». Помимо прочего, данный способ позволяет разделить один объект между несколькими другими.

Неплохой пример приведен в википедии:

[https://ru.wikipedia.org/wiki/Агрегирование\\_\(программирование\)](https://ru.wikipedia.org/wiki/Агрегирование_(программирование))

В рамках лабораторной работы требуется вывести на экран несколько графических объектов (чайников), каждый из которых обладает определенным материалом, характеризующим его цвет. При этом материалы могут повторяться, например, можно вывести два красных глянцевых чайника в разных местах сцены. Тем не менее, даже в этом случае, одно остается неизменным — каждый графический объект имеет материал, то есть материал является неотъемлемой частью графического объекта. Таким образом, можно говорить, что графический объект состоит из следующих частей:

- Позиции и ориентации объекта, выраженной матрицей модели. Поскольку матрица модели не имеет смысла без графического объекта и нет необходимости делить матрицу модели между несколькими графическими объектами, связь между графическим объектом и матрицей модели выполнена по принципу композиции. В этом случае графический объект в качестве одного из своих полей имеет объект типа `GLfloat[16]`.
- Материала, который определяется с помощью класса `PhongMaterial`, реализованного ранее. Поскольку материал может создаваться независимо от графического объекта и несколько графических объектов могут использовать один и тот же материал, то связь между двумя классами выполнена по принципу агрегации. В этом случае графический объект в качестве одного из своих полей имеет указатель на объект типа `PhongMaterial`.
- Геометрической формы, которая пока задается с помощью функции `glutSolidTeapot`. Для представления графической формы отдельный класс пока не реализован.

## Агрегация.

Для указания того, что каждому графическому объекту соответствует некоторый материал, используется понятие агрегации. В объектно-ориентированном программировании под агрегированием подразумевают методику создания нового класса из уже существующих классов путём их включения.

Для реализации этого отношения в класс для работы с графическим объектом требуется внести определенные изменения, которые бы позволяли хранить указатель на используемый материал, а также инициализировать значение этого указателя.

Измененная структура класса GraphicObject представлена ниже:

```
// КЛАСС ДЛЯ ПРЕДСТАВЛЕНИЯ ОДНОГО ГРАФИЧЕСКОГО ОБЪЕКТА
class GraphicObject
{
public:
    // Конструктор
    GraphicObject();

    // Установка и получение позиции объекта
    void setPosition(vec3 position);
    vec3 getPosition();

    // Установка и получения угла поворота в градусах
    // поворот осуществляется в горизонтальной плоскости
    // вокруг оси Oy по часовой стрелке
    void setAngle(float grad);
    float getAngle();

    // Установка текущего цвета объекта
    void setColor(vec3 color);
    vec3 getColor();

    // Установка используемого материала
    void setMaterial(PhongMaterial *material);

    // Вывести объект
    void draw();

private:
    // Позиция объекта в глобальной системе координат
    vec3 position;
    // Угол поворота в горизонтальной плоскости (в градусах)
    float angle;
    // Цвет модели
    vec3 color;
    // Матрица модели (расположение объекта) - чтоб не вычислять каждый раз
    GLfloat modelMatrix[16];
    // Используемый материал
    PhongMaterial *material;

private:
    // расчет матрицы modelMatrix на основе position и angle
    void recalculateModelMatrix();
};
```

В класс был добавлен указатель на используемый материал. Данное поле указывает на объект типа Material. При этом если несколько разных графических объектов используют один и тот же материал, они могут указывать на один и тот же объект. Для установки этого приватного поля используется метод setMaterial.



## Делегирование.

На базе агрегирования или композиции реализуется методика делегирования, когда поставленная перед внешним объектом задача перепоручается внутреннему объекту, специализирующемуся на решении задач такого рода.

Рассмотрим процесс построения изображения. При необходимости вывода сцены на экран вызывается функция `display`, которая знает о существовании камеры, источника света и нескольких графических моделей. Не вдаваясь в подробности того, как все это реализовано, данная функция вызывает необходимые методы каждого объекта в определенной последовательности. В результате функция `display` выглядит следующим образом:

```
// CALLBACK-ФУНКЦИЯ ВЫЗЫВАЕТСЯ ПРИ ПЕРЕРИСКОВКЕ ОКНА
void display()
{
    // очищаем буфер цвета и буфер глубины
    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);
    // включаем режим расчета освещения
    glEnable(GL_LIGHTING);

    // устанавливаем камеру
    camera.apply();

    // устанавливаем источник света
    light.apply(GL_LIGHT0);

    // выводим все графические объекты
    for (auto& go : graphicObjects) {
        go.draw();
    }

    // смена переднего и заднего буферов
    glutSwapBuffers();

    // выводим заголовок окна
    char windowTitle[128];
    int FPS = getFps();
    sprintf_s(windowTitle, 128, "Laba_05 [%i FPS]", FPS);
    glutSetWindowTitle(windowTitle);
};
```

Функция `display` для вывода каждого графического объекта вызывает его метод `draw` (метод класса `GraphicObject`), полагая, что графический объект сам знает, как его выводить. В результате функция `display` делегирует задачу по выводу графического объекта самому графическому объекту.

При этом метод `draw` класса `GraphicObject`, в свою очередь, выглядит следующим образом:

```
// вывести объект
void GraphicObject::draw()
{
    glColor3f(color.r, color.g, color.b);

    if (material != nullptr) {
        material->apply();
    }

    glPushMatrix();
    glMultMatrixf(modelMatrix);
    glutSolidTeapot(1.0);
    glPopMatrix();
}
```



Метод draw класса GraphicObject также выполняет все, что может, но некоторые задачи перепоручает классу, который он агрегирует, в частности, классу для представления материала - PhongMaterial. Класс для работы с материалом, в свою очередь, выполняет все, что требуется, для установки параметров материала:

```
// применение материала
void PhongMaterial::apply(void)
{
    ...
}
```

Данный подход имеет множество преимуществ, начиная от упрощения разработки программы, поскольку при написании классов можно сосредоточиться на написании одной функциональности и отложить решение других проблем на потом, и заканчивая возможностью изменять реализацию одних классов, не затрагивая другие.

Понятия композиция, агрегация и делегирование используются повсеместно, при реализации любых программ с использованием любых языков программирования, поэтому жизненно важно знать, что это такое и уметь применять на практике.

### **Недостаток использования «сырых» указателей (raw pointers).**

Приведенная выше структура класса GraphicObject является почти удовлетворительной за исключением одной маленькой детали - использование обычных указателей (raw pointers) для организации агрегации имеет определенные недостатки.

Если предположить, что графические объекты будут создаваться и уничтожаться динамически, то возникает вопрос, что делать с материалом при уничтожении графического объекта? В этом случае возможны несколько вариантов:

- При уничтожении графического объекта в его деструкторе будет также удаляться и связанный с ним материал (принудительно вызываться оператор delete). При этом в случае, если этот материал использовался в каком-либо другом объекте, то он станет недоступен для того объекта и программа, скорее всего, завершится аварийно (обращение к недоступной области памяти).
- При уничтожении графического объекта связанный с ним материал не удаляется. В этом случае, может возникнуть ситуация, когда какой-либо материал больше не используется и на него нет указателей. В результате он будет существовать до конца работы программы, занимая ценную память (утечка памяти). Если объекты создаются и уничтожаются довольно часто, то рано или поздно вся память будет переполнена, и программа будет закрыта аварийно.

Для решения этой проблемы в лабораторной работе необходимо использовать умные указатели (smart pointers), описание которых приведено в следующем разделе.

## Использование умных указателей (Smart Pointers).

Указатели в C++ являются необычайно мощным средством для написания эффективных алгоритмов, которые могут создавать, хранить, передавать и обрабатывать большие объемы данных с минимальными накладными расходами. Тем не менее, использование указателей требует необычайной осторожности, поскольку их некорректное использование может привести к катастрофическим последствиям.

**Одной из важнейших проблем является утечка памяти (memory leak).** Если функция в процессе своей работы выделяет динамическую память, необходимо гарантировать, что данная память будет освобождена при завершении функции. То есть функция должна быть написана так, чтобы при любых вариантах её использования (любых ветках исполнения) выполнялся код по освобождению ранее выделенной памяти. Если функция забывает удалить хотя бы один байт динамически выделенной памяти и при этом вызывается достаточно часто, то это рано или поздно приведет к тому, что вся свободная память будет израсходована и программа будет завершена аварийно. Проблема усугубляется возможностью срабатывания исключений, в результате чего выполнение функции прерывается, управление передается обработчику исключений, а код, реализующий освобождение памяти, никогда не будет выполнен.

**Другой существенной проблемой является определение владельца выделенной памяти.** Предположим, что существует функция, динамически создающая какой-либо объект и возвращающая указатель на него. Возникает вопрос, кто должен освобождать эту память: код, который получил данный указатель, или библиотека, которой принадлежит функция, создающая объект? Более того, зачастую, динамически созданный в памяти ресурс может совместно использоваться несколькими участками кода и необходимо определить, кто будет его удалять и когда это должно произойти. Если память вообще не будет удалена, это приведет к ранее описанной проблеме утечки памяти. Если память будет удалена слишком рано, когда она еще используется какими-либо другими участками кода, то это приведет к абсолютно непредсказуемым ошибкам.

Использование указателей порождает и другие проблемы, среди которых:

- Память, выделенная с помощью оператора `new`, должна быть удалена с помощью оператора `delete`, а память, выделенная с помощью оператора `new[]`, должна быть удалена с использованием оператора `delete[]`. Использование неверного оператора удаления приводит к трудно-вылавливаемым и очень неприятным ошибкам;
- Нельзя пытаться удалить статически выделенную память. При этом не всегда бывает понятно, возвращает ли некоторая функция указатель на динамически выделенную память, или адрес некоторой области, выделенной статически;
- Не следует повторно освобождать память, которая уже была освобождена, особенно если дело касается динамически создаваемых массивов объектов, а не одного объекта.

Для решения всех вышеперечисленных проблем в C++ используются умные указатели (Smart Pointers). Умные указатели – это шаблонные классы, являющиеся обертками над обычными указателями. Основной целью использования умных указателей является инкапсуляция работы с динамической памятью внутри этих классов таким образом, чтобы свойства и поведение умных указателей имитировали свойства и поведение обычных указателей. При этом на умные указатели возлагается обязанность своевременного и корректного освобождения выделенных ресурсов в тот момент, когда это является необходимым. Использование умных указателей упрощает разработку кода и процесс отладки, исключая утечки памяти и её преждевременное удаление.

В C++ различают три вида умных указателей:

- **Уникальный указатель (`std::unique_ptr`).** Данный вид указателя используется, когда необходимо, чтобы у памяти был только один владелец – переменная типа `unique_ptr`. Уникальные указатели не могут быть скопированы, поэтому гарантируется, что на определенную динамически выделенную память указывает только один указатель. Как только программа покинет область видимости переменной типа `unique_ptr`, автоматически и гарантированно вызовется деструктор этой переменной и память, связанная с ней, будет удалена.

- **Разделяемый указатель (`std::shared_ptr`).** Разделяемые указатели – это умные указатели, которые позволяют совместно использовать общую, динамически выделяемую память, то есть несколько разделяемых указателей указывают на одну и ту же область памяти. При этом память будет освобождена только тогда, когда не останется ни одного указателя, указывающего на неё.
- **Слабый указатель (`std::weak_ptr`).** Слабые указатели – это умные указатели, которые содержат указатель на память, но не являются её владельцем. Слабые указатели используются для решения проблем циклических ссылок, когда два объекта ссылаются друг на друга и поэтому не могут быть удалены, несмотря на то, что нет ни одного внешнего указателя на эти объекты:



Далее будут представлены некоторые сведения и примеры работы с умными указателями. Для лучшего понимания рекомендуется самостоятельно набрать и запустить все примеры. Кроме того, рекомендуется также просмотреть видео-уроки, доступные по следующим ссылкам:

<https://www.youtube.com/watch?v=ixsTu-ULh0Q>  
<https://www.youtube.com/watch?v=dpRozfXepnA>  
<https://www.youtube.com/watch?v=edGrIXZJEA>

Для примеров, показывающих работу с умными указателями, будет использоваться класс, представленный ниже. Данный класс необходим исключительно для демонстрации того, в какой момент происходит создание и удаление объекта, на который указывает умный указатель:

```

// ДЕМО-КЛАСС ДЛЯ ПРОВЕРКИ ВЫЗОВОВ КОНСТРУКТОРОВ И ДЕКТРУКТОРОВ
class Entity
{
public:
    // конструктор
    Entity(string name) {
        this->name = name;
        cout << "Constructor for Entity " << name << endl;
    }

    // деструктор
    ~Entity() {
        cout << "~Destructor for Entity " << name << endl;
    }

    // метод класса, который выполняет какую-то работу
    void doSomething() {
        cout << "Doing something with Entity " << name << endl;
    }

private:
    // имя объекта - для того чтобы различать их
    string name;
};

```

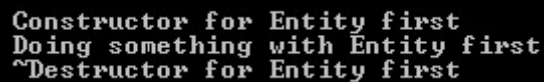
Ниже представлен пример функции, использующей уникальный указатель для управления динамически создаваемым объектом.

```
// ФУНКЦИЯ, ИСПОЛЬЗУЮЩАЯ УМНЫЙ УКАЗАТЕЛЬ
void foo() {
    // создаем умный указатель, который будет указывать на объект типа Entity
    // объект создается непосредственно при создании умного указателя
    // (является параметром конструктора)
    unique_ptr<Entity> pointer(new Entity("first"));

    // используем умный указатель произвольным образом, также
    // как использовался бы обычный указатель на Entity
    pointer->doSomething();

    // самостоятельное удаление памяти не требуется,
    // память будет удалена в деструкторе умного указателя после того,
    // как программа покинет область видимости этого умного указателя
    return;
}
```

Результат работы данного примера, то есть выводимая на консоль информация, приведена на следующем рисунке. Наличие диагностических сообщений позволяет убедиться, что объект действительно был удален без непосредственного вызова оператора delete:



```
Constructor for Entity first
Doing something with Entity first
~Destructor for Entity first
```

### Подключение библиотеки memory.

Для работы с умными указателями необходимо подключить библиотеку <memory>, которая является частью стандартной библиотеки C++. Все типы данных библиотеки объявлены в пространстве имен std, поэтому для облегчения оперирования соответствующими типами данных можно указать используемое по умолчанию пространство имен:

```
#include <memory>
...

using namespace std;
...
```

### Использование уникальных указателей (unique\_ptr).

Уникальный указатель – это умный указатель, который является владельцем динамически выделенной памяти и управляет ею, в частности, освобождает эту память, если она больше не нужна. Особенностью уникального указателя является то, что он не может быть скопирован, поэтому можно считать, что на одну область памяти указывает только один уникальный указатель. Как только программа покидает область видимости этого указателя, динамически выделенная память автоматически освобождается.

Прежде всего, необходимо рассмотреть процесс создания уникального указателя и его инициализацию. Уникальный указатель является шаблонным классом, поэтому при объявлении переменной типа unique\_ptr необходимо указать тип данных, на который будет указывать этот указатель:

```
// создаем "пустой" уникальный указатель,
// то есть указатель, который никуда не указывает
unique_ptr<Entity> pointer = nullptr;
```

Необходимо отметить, что при создании «пустых» указателей рекомендуется явным образом «обнулять» их, то есть присваивать им значение равное nullptr.

Пустой указатель не является особо полезным, поэтому необходимо рассмотреть способы инициализации уникального указателя так, чтобы он указывал на динамически выделяемую память. Сделать это можно двумя способами:

```
// создаем два уникальных указателя:  
// первый указывает на динамически созданный объект с именем "first"  
unique_ptr<Entity> pointer_1(new Entity("first"));  
// второй указывает на динамически созданный объект с именем "second"  
unique_ptr<Entity> pointer_2 = make_unique<Entity>("second");
```

В первом случае объект типа Entity создается явно с помощью оператора new и результат работы этого оператора (то есть адрес памяти) передается в конструктор умного указателя. Начиная с этого момента данный указатель является владельцем памяти и она будет автоматически удалена, как только программа выйдет из области видимости переменной pointer\_1.

Во втором случае для инициализации умного указателя используется шаблонная функция make\_unique, которая самостоятельно создает новый объект указанного типа(Entity), вызывает его конструктор с переданными параметрами ("second") и инициализирует умный указатель. Второй способ, несмотря на то, что он является более громоздким, является предпочтительным, поскольку позволяет избежать некоторых довольно специфичных проблем.

После того, как уникальный объект инициализирован, его можно использовать как обычный указатель за счет того, что в классе unique\_ptr естественным образом переопределены операторы (\*) и (->):

```
// использование умных указателей для доступа к Entity  
pointer_1->doSomething();  
(*pointer_1).doSomething();
```

Иногда возникает необходимость получить обычный указатель на ту область памяти, куда указывает умный указатель, например, для передачи в legacy-код, то есть в код, который был написан без использования умных указателей и который в качестве параметра принимает обычные C++ указатели. Для этого используется метод get(), однако ни в коем случае не следует освобождать память, на которую указывает данный указатель, самостоятельно:

```
// получение raw-указателя для его использования  
// или передачи в какую-либо legacy-функцию  
Entity* p = pointer_1.get();  
p->doSomething();  
someLegacyFunc(p);
```

Также иногда требуется повторно инициализировать ранее созданный указатель так, чтобы он указывал на новую область памяти. При этом старая область памяти (поскольку совершенно очевидно, что она уже не нужна) будет автоматически удалена. Следует отметить, что нельзя просто присвоить указателю адрес новой памяти или другой уникальный указатель (в этом и заключается его особенность), вместо этого используется функция reset() или функция make\_unique:

```
// инициализация указателя другим значением  
// (предыдущая область памяти будет удалена)  
pointer_1.reset(new Entity("third"));  
pointer_1 = make_unique<Entity>("forth");
```

Несмотря на то, что удаление связанной с умным указателем памяти происходит автоматически, как только программа покидает область видимости уникального указателя, иногда это необходимо сделать самостоятельно. Например, в случае если в ранее выделенной памяти больше нет необходимости, а она имеет большой объем. Для освобождения памяти можно присвоить указателю нулевое значение или вызвать функцию reset без параметров:

```
// самостоятельное освобождение памяти, если в ней больше нет необходимости  
pointer_1 = nullptr;  
pointer_2.reset();
```

Также можно выполнить проверку на то, что уникальный указатель не является пустым, то есть указывает на какую-либо память, для того чтобы случайно не обратиться по нулевому адресу:

```
// проверка на то, что указатель не "пустой"
if (pointer_1 != nullptr) {
    pointer_1->doSomething();
}
```

Последняя функция, на которую следует обратить внимание – это функция `release()`. Данная функция позволяет получить адрес памяти, которой управляет уникальный указатель и одновременно отказать от его управления. То есть теперь умный указатель является пустым, а забота об удалении соответствующего участка памяти перекладывается на код, который вызвал функцию `release()`.

В завершении раздела, посвященного умному указателю `unique_ptr`, приводится текст функции, которая демонстрирует некоторые методы работы с ним:

```
// ФУНКЦИЯ, ИСПОЛЬЗУЮЩАЯ УМНЫЙ УКАЗАТЕЛЬ
void foo() {
    // создаем уникальный указателя на объект типа Entity("First")
    unique_ptr<Entity> pointer = make_unique<Entity>("First");

    // выполняем определенные действия с этим объектом
    pointer->doSomething();

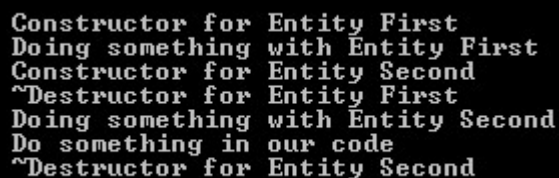
    // переинициализируем указатель:
    // создается Entity("Second"), а Entity("First") автоматически удаляется
    pointer = make_unique<Entity>("Second");

    // выполняем определенные действия
    // при этом действия выполняются уже с объектом Entity("Second")
    pointer->doSomething();

    // явным образом объект не удаляется
    // выполняется какая-то другая работа
    cout << "Do something in our code" << endl;

    // объект автоматически удаляется при завершении функции, то есть в тот момент,
    // когда программа выйдет из области видимости переменной pointer
    // оператор return здесь не обязателен и служит для обозначения конца функции
    return;
}
```

Далее приводится содержимое консоли, которое позволяет лучше понять, в какие моменты происходит создание и удаление объектов типа `Entity`:



```
Constructor for Entity First
Doing something with Entity First
Constructor for Entity Second
~Destructor for Entity First
Doing something with Entity Second
Do something in our code
~Destructor for Entity Second
```

Резюмируя можно отметить, что уникальный указатель (`unique_ptr`) используется, когда требуется гарантированно удалять выделенную динамическую память, и при этом данная память не будет использоваться из нескольких разных функций, то есть можно считать, что у такой памяти есть только один владелец – это функция, которая выделила и использует эту память.

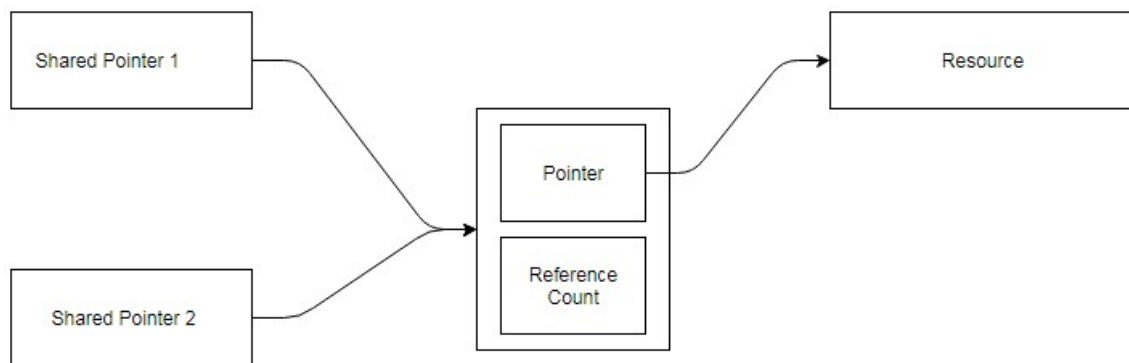
Следует отметить, что здесь приведены далеко не все сведения, касающиеся уникального указателя. Например, уникальный указатель поддерживает `move`-семантику, позволяет управлять не одним объектом, а массивом объектов, а также позволяет указать пользовательскую функцию удаления (`associated deleter`).

## Использование разделяемых указателей (shared\_ptr).

Разделяемый указатель (shared\_ptr) реализует другую концепцию, нежели уникальный указатель. Он позволяет присваивать одному разделяемому указателю, значение другого разделяемого указателя, приводя, таким образом, к тому, что два указателя будут указывать на одну и ту же динамически выделенную память, то есть будут делить общий ресурс. Работать с данной памятью можно будет одновременно через оба указателя и удалена она будет только после того, как не останется ни одного указателя, который бы указывал на неё.

Для определения того, сколько существует указателей на динамически выделенную память используется техника, которая получила название «подсчёт ссылок» (reference counting). Каждый раз, когда с помощью разделяемого указателя выделяется новая область памяти (ресурс), количество ссылок устанавливается равным единице, что сигнализирует о том, что существует один разделяемый указатель, который указывает на только что выделенную область памяти.

Далее, если происходит копирование разделяемого указателя, количество ссылок увеличивается на единицу, сигнализируя, таким образом, о том, что на ту же область памяти ссылается несколько указателей. Схематично указанная взаимосвязь и необходимая внутренняя структура данных приведены на следующем рисунке:



При удалении разделяемого указателя, например, если программа покидает область видимости этой переменной, происходит уменьшение количества ссылок на единицу. Если количество ссылок уменьшилось до нуля, то есть на динамически выделенную память больше не указывает ни один разделяемый указатель, происходит освобождение памяти.

Таким образом, если работать с динамически выделяемой памятью только через разделяемые указатели, можно гарантировать, что память не будет удалена до тех пор, пока есть хотя бы одна ссылка на неё (то есть, существует хотя бы один разделяемый указатель). Вместе с тем, также можно гарантировать, что область памяти будет освобождена, если таких указателей нет. Всё это делает разделяемые указатели весьма полезными при передаче указателей на память в качестве параметров функции или возврата из функций указателей на динамически созданные объекты.

Работа с разделяемыми указателями мало чем отличается от работы с уникальными указателями. Прежде всего, необходимо рассмотреть процесс создания и инициализации разделяемых указателей. Следует отметить, что второй способ (с использованием функции make\_shared) является предпочтительным.

```
// создаем разделяемый указатель
shared_ptr<Entity> pointer_1(new Entity("First"));
shared_ptr<Entity> pointer_2 = make_shared<Entity>("Second");
```

Далее приводится процесс копирования ссылок, что позволяет организовать совместное владение динамически выделенной памятью:

```
// копирование указателей
// объект Entity("Second") удаляется, поскольку на него не осталось ссылок
// обе переменные указывают на объект Entity("First")
pointer_2 = pointer_1;
// для проверки можно вывести количество активных ссылок на разделяемый ресурс
cout << "Reference count = " << pointer_1.use_count() << endl;
```



Следует обратить внимание, что теперь оба указателя ссылаются на одну и ту же область памяти, поэтому если область памяти будет изменена через один указатель, второй указатель тоже заметит эти изменения, то есть происходит копирование именно указателей (адресов памяти), а не содержимого памяти.

Работа с памятью через разделяемый указатель ничем не отличается от работы с памятью через уникальный указатель. Единственное, что следует иметь в виду, это то, что освобождение памяти происходит только тогда, когда обе ссылки перестанут указывать на эту память.

Пример функции, иллюстрирующей работу с указателями типа `shared_ptr`, представлен ниже:

```
// ФУНКЦИЯ, ИСПОЛЬЗУЮЩАЯ УМНЫЙ УКАЗАТЕЛЬ
void foo() {
    // создаем разделяемый указатель
    shared_ptr<Entity> pointer_1(new Entity("First"));
    shared_ptr<Entity> pointer_2 = make_shared<Entity>("Second");

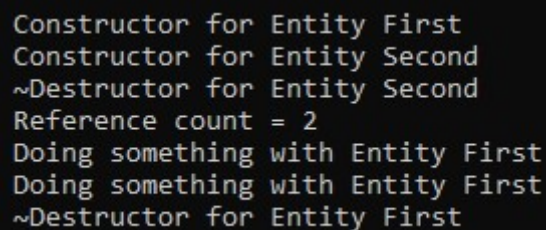
    // копирование указателей
    // объект Entity("Second") удаляется, поскольку на него не осталось ссылок
    // обе переменные указывают на объект Entity("First")
    pointer_2 = pointer_1;
    // для проверки можно вывести количество активных ссылок на разделяемый ресурс
    cout << "Reference count = " << pointer_1.use_count() << endl;

    // обращаемся к памяти через первый разделяемый указатель
    if (pointer_1 != nullptr) {
        pointer_1->doSomething();
    };
    //обнуляем первый указатель (объект все еще существует)
    pointer_1 = nullptr;

    // обращаемся к памяти через второй разделяемый указатель
    if (pointer_2 != nullptr) {
        pointer_2->doSomething();
    };
    //обнуляем второй указатель (объект удаляется)
    pointer_2 = nullptr;

    // оператор return здесь не обязателен и служит для обозначения конца функции
    return;
}
```

Далее приводится содержимое консоли, которое позволяет лучше понять, в какие моменты происходит создание и удаление объектов типа `Entity`:



```
Constructor for Entity First
Constructor for Entity Second
~Destructor for Entity Second
Reference count = 2
Doing something with Entity First
Doing something with Entity First
~Destructor for Entity First
```

Класс для работы с разделяемыми указателями имеет также и другие методы, в частности, ранее рассмотренные методы `get()` и `reset()`, а так же довольно интересные методы `use_count()` и `unique()`.

Отдельно следует отметить, что совместное владение памятью несколькими указателями типа `shared_ptr` возможно только в том случае, если новые указатели были созданы с помощью конструктора копирования или оператора присваивания (в том числе при передаче в функцию и из функции). Создание новых разделяемых указателей путем передачи в качестве параметра raw-указателя (полученного с помощью метода `get()`) приводит к непредсказуемым последствиям.



## Использование слабых указателей (weak\_ptr).

Слабый указатель – это умный указатель, который позволяет ссылаться на ресурс (динамически выделенную память), но не является его владельцем. Слабые указатели используются совместно с разделяемыми указателями. В этом случае память не освобождается до тех пор, пока на неё есть хотя бы один разделяемый указатель. Наличие слабых указателей, указывающих на ту же самую память, не сказывается на возможности её освобождения, то есть память может быть освобождена, даже если на неё есть слабые указатели.

Использование слабых указателей необходимо, чтобы избежать проблем перекрестного или циклического связывания. Например, при реализации двухсвязного списка существуют внешние указатели, которые указывают на начало и конец списка, а так же внутренние указатели, которые связывают элементы списка друг с другом:



В случае если в списке больше нет необходимости, можно удалить указатели на начало и конец, что сигнализирует о том, что данная память больше не нужна, и она может быть освобождена. Однако в случае использования разделяемых указателей это не произойдет, поскольку элементы списка будут ссылаться друг на друга. Для решения этой проблемы в качестве указателей на начало и конец списка по-прежнему используются разделяемые указатели (shared\_ptr), а для связи элементов списка друг с другом слабые указатели (weak\_ptr). В этом случае, как только пропадет последний разделяемый указатель, память будет освобождена, даже если существуют слабые указатели, что, в принципе, является логичным.

Слабые указатели используются достаточно редко и в данной лабораторной не применяются, тем не менее, для полного охвата темы, далее рассматривается процесс создания и инициализации слабых указателей.

Слабые указатели неразрывно связаны с разделяемыми указателями, то есть они могут указывать только на ту области памяти, на которую указывают разделяемые указатели и поэтому инициализируются на основе ранее созданных и инициализированных разделяемых указателей:

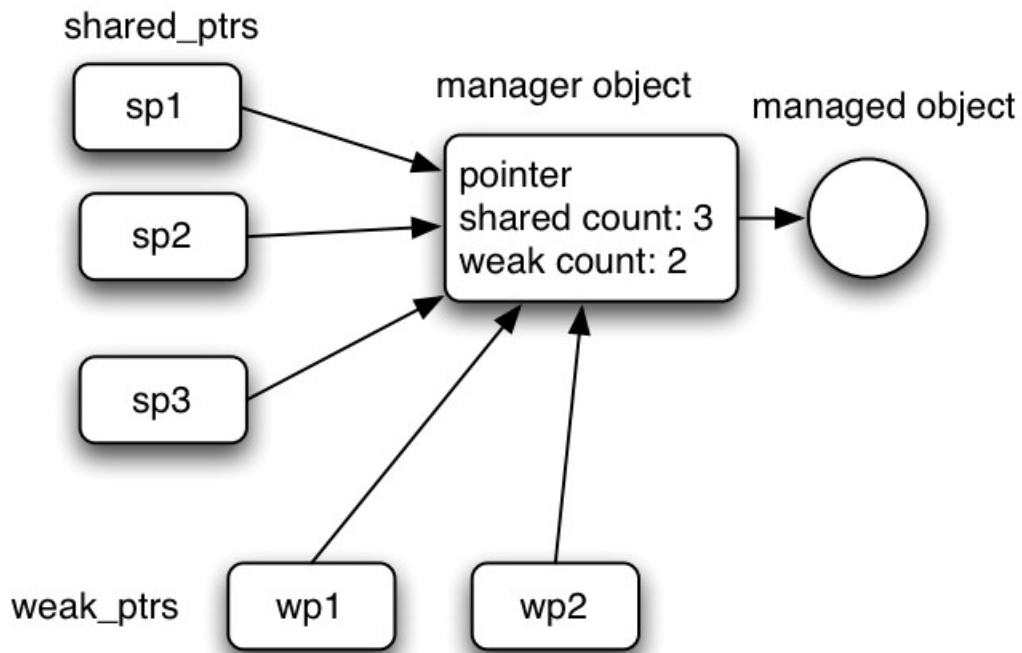
```
// создаем разделяемый указатель
shared_ptr<Entity> pointer(new Entity("First"));
// создаем пустой слабый указатель
weak_ptr<Entity> weak_pointer_1;
// инициализируем пустой указатель
// теперь он указывает на ту же область памяти, что и разделяемый указатель
weak_pointer_1 = pointer;
```

Слабые указатели можно так же инициализировать непосредственно при создании, передавая в качестве параметра конструктора разделяемый указатель:

```
// создаем еще один слабый указатель на основе разделяемого указателя
weak_ptr<Entity> weak_pointer_2(pointer);
```

В момент инициализации слабых указателей он присоединяется к группе разделяемых указателей, которые совместно используют динамически выделенную память. Если все разделяемые указатели прекратят своё существование, то сама память будет освобождена, но, тем не менее, группа останется и каждый слабый указатель будет помнить, что он ссылался на какой-либо ресурс, но теперь этого ресурса больше нет.

Ниже приведен рисунок, который показывает взаимосвязь слабых и разделяемых указателей, а также вспомогательную структуру данных, необходимую для реализации этой функциональности:



Для проверки того, существует ли еще ресурс, на который указывает слабый указатель (то есть, остался ли еще хотя бы один разделяемый указатель в той же группе, что и выбранный слабый указатель), можно воспользоваться специальным методом `expired()`, который возвращает `true`, если связанный с ним объект уже был удален.

Вышеприведенный метод не является обязательным к использованию, поскольку для того, чтобы обратиться к памяти, на которую указывает слабый указатель, обязательно требуется преобразовать его к разделяемому указателю с помощью метода `lock()`. Метод `lock()` создает и возвращает еще один разделяемый указатель, связанный с тем же ресурсом, что и слабый указатель. Это гарантирует, что за время использования полученного указателя, объект не будет удален. Если же, к моменту вызова функции `lock()` ресурс уже был удален, то будет возвращен нулевой указатель:

```
// доступ к памяти через слабый указатель
// доступ напрямую невозможен, требуется получить разделяемый указатель
shared_ptr<Entity> temp = weak_pointer_1.lock();
// используя полученный указатель, работаем с объектом
if (temp != nullptr) {
    temp->doSomething();
};
// после выполнения действий обнуляем временный разделяемый указатель
temp = nullptr;
```

## Определение связи между графическим объектом и материалом (v 2.0).

Для формирования окончательной структуры класса `GraphicObject` необходимо объединить агрегацию с использованием умных указателей, в данном случае разделяемых указателей `shared_ptr`. В итоге структура класса `GraphicObject` будет выглядеть следующим образом:

```
// КЛАСС ДЛЯ ПРЕДСТАВЛЕНИЯ ОДНОГО ГРАФИЧЕСКОГО ОБЪЕКТА
class GraphicObject
{
public:
    // Конструктор
    GraphicObject();

    // Установка и получение позиции объекта
    void setPosition(vec3 position);
    vec3 getPosition();

    // Установка и получения угла поворота в градусах
    // поворот осуществляется в горизонтальной плоскости
    // вокруг оси Oy по часовой стрелке
    void setAngle(float grad);
    float getAngle();

    // Установка текущего цвета объекта
    void setColor(vec3 color);
    vec3 getColor();

    // Установка используемого материала
    void setMaterial(std::shared_ptr<PhongMaterial> material);

    // Вывести объект
    void draw();

private:
    // Позиция объекта в глобальной системе координат
    vec3 position;
    // Угол поворота в горизонтальной плоскости (в градусах)
    float angle;
    // Цвет модели
    vec3 color;
    // Матрица модели (расположение объекта) - чтоб не вычислять каждый раз
    GLfloat modelMatrix[16];
    // Используемый материал
    std::shared_ptr<PhongMaterial> material;

private:
    // расчет матрицы modelMatrix на основе position и angle
    void recalculateModelMatrix();
};
```

При этом для хранения всех материалов, поскольку заранее неизвестно сколько их будет в последующих лабораторных работах, можно использовать вектор (`std::vector`) разделяемых указателей (`std::shared_ptr`). Данный вектор может быть объявлен следующим образом:

```
// используемые материалы
std::vector<std::shared_ptr<PhongMaterial>> materials;
```

На первый взгляд данная конструкция может казаться чересчур громоздкой и непонятной, но при определенном опыте не должна вызывать никаких проблем. Умение читать подобный код и писать его самостоятельно – одна из основных целей данного курса лабораторных работ.

## Задание к лабораторной работе.

Лабораторная работа №5 строится на основе предыдущей работы с внесением необходимых изменений. При этом к лабораторной работе предъявляются следующие требования:

1. Для работы с источником света необходимо создать отдельный класс `Light`, структура которого приведена в соответствующем разделе. Класс должен быть оформлен в виде отдельного модуля.
2. Для работы с материалом необходимо создать класс `PhongMaterial`, структура которого приведена в соответствующем разделе. Класс должен быть оформлен в виде отдельного модуля.
3. В классе `PhongMaterial` должен быть реализован метод `load`, позволяющий загружать параметры материала из внешнего файла, пример которого доступен в демонстрационной программе.
4. Необходимо организовать связь между классами `GraphicObject` и `PhongMaterial` используя агрегацию и разделяемый умный указатель (`shared_ptr`) в соответствии с примером, приведенном в разделе «определение связи между графическим объектом и материалом (v 2.0)».
5. Для демонстрации необходимо вывести на экран четыре чайник с материалами, взятыми из примера к методическим указаниям.

Отчет к лабораторной работе должен быть сформирован на основе шаблона, представленного на следующей странице. При оформлении отчета необходимо внести свою фамилию, имя, отчество, номер группы, а также заполнить таблицу. При заполнении таблицы необходимо дать краткое описание реализованным модулям, указать количество строк каждого модуля, а также подсчитать общее количество строк во всем проекте.

## Алгоритмические основы компьютерной графики (5 семестр)

Выполнил: **Иванов Иван Иванович**

Группа: 00-00

Проверил: **Галибин Сергей Владимирович**

### Задание:

Лабораторная работа №5 строится на основе предыдущей работы с внесением необходимых изменений. При этом к лабораторной работе предъявляются следующие требования:

1. Для работы с источником света необходимо создать отдельный класс `Light`, структура которого приведена в соответствующем разделе. Класс должен быть оформлен в виде отдельного модуля.
2. Для работы с материалом необходимо создать класс `PhongMaterial`, структура которого приведена в соответствующем разделе. Класс должен быть оформлен в виде отдельного модуля.
3. В классе `PhongMaterial` должен быть реализован метод `load`, позволяющий загружать параметры материала из внешнего файла, пример которого доступен в демонстрационной программе.
4. Необходимо организовать связь между классами `GraphicObject` и `PhongMaterial` используя агрегацию и разделяемый умный указатель (`shared_ptr`) в соответствии с примером, приведенном в разделе «определение связи между графическим объектом и материалом (v 2.0)».
5. Для демонстрации необходимо вывести на экран четыре чайника с материалами, взятыми из примера к методическим указаниям.

### Состав проекта:

Название модуля и его назначение	Количество строк	
	*.h	*.cpp
main - основная программа		
Data -		
Display -		
Simulation -		
GraphicObject -		
Camera -		
Light -		
PhongMaterial -		