

Министерство образования и науки Российской Федерации

Калужский филиал  
федерального государственного бюджетного образовательного  
учреждения высшего образования  
**«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»**  
(КФ МГТУ им. Н.Э. Баумана)

**Ю.С. Белов**

## **ВЫВОД РАСТРОВЫХ ИЗОБРАЖЕНИЙ В OPENGL**

Методические указания к лабораторной работе  
по дисциплине «Компьютерная графика»

Калуга, 2018

УДК 004.62  
ББК 32.972.5  
Б435

Методические указания составлены в соответствии с учебным планом КФ МГТУ им. Н.Э.Баумана по направлению подготовки 09.03.04 «Программная инженерия» кафедры «Программного обеспечения ЭВМ, информационных технологий и прикладной математики».

Методические указания рассмотрены и одобрены:

- Кафедрой «Программного обеспечения ЭВМ, информационных технологий и прикладной математики» (ФН1-КФ) протокол № 7 от «21» февраля 2018 г.

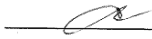
И.о. зав. кафедрой ФН1-КФ  к.т.н., доцент Ю.Е. Гагарин

- Методической комиссией факультета ФНК протокол № 2 от «26» 02 2018 г.

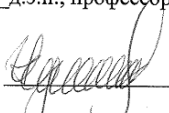
Председатель методической комиссии факультета ФНК  к.х.н., доцент К.Л. Анфилов

- Методической комиссией КФ МГТУ им.Н.Э. Баумана протокол № 2 от «06» 03 2018 г.

Председатель методической комиссии КФ МГТУ им.Н.Э. Баумана

 д.э.н., профессор О.Л. Перерва

Рецензент:  
к.т.н., зав. кафедрой ЭИУ2-КФ

 И.В. Чухраев

Авторы  
к.ф.-м.н., доцент кафедры ФН1-КФ

 Ю.С. Белов

#### Аннотация

Методические указания по выполнению лабораторной работы по курсу «Компьютерная графика» содержат общие сведения о применении программного интерфейса OpenGL для работы с растровыми изображениями. В методических указаниях приводятся теоретические сведения о битовых и растровых масках и способах их оптимизации реализуемых в OpenGL. Рассмотрен процесс передачи отдельных пикселей преобразования в пространственном и цветовом отношениях, а также способы упаковки.

Предназначены для студентов 2-го курса бакалавриата КФ МГТУ им. Н.Э. Баумана, обучающихся по направлению подготовки 09.03.04 «Программная инженерия».

© Калужский филиал МГТУ им. Н.Э. Баумана, 2018 г.  
© Ю.С. Белов, 2018 г.

## ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ .....	4
ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ .....	5
ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ .....	6
ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ .....	19
ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ .....	46
ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ .....	46
ВАРИАНТЫ ЗАДАНИЙ .....	46
КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ .....	50
ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ .....	51
ОСНОВНАЯ ЛИТЕРАТУРА .....	52
ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА.....	53

## ВВЕДЕНИЕ

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Компьютерная графика» на кафедре «Программное обеспечение ЭВМ, информационные технологии и прикладная математика» факультета фундаментальных наук Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания, ориентированные на студентов 2-го курса бакалавриата направления подготовки 09.03.04 «Программная инженерия», содержат краткую теоретическую часть, описывающую способы представления растровых масок средствами программного интерфейса OpenGL, поэтапные примеры создания обработки отдельных пикселей и их изменения в выделенных системах координат, комментарии и пояснения по вышеназванным этапам, а также задание на лабораторную работу.

Методические указания составлены в расчете на начальное ознакомление студентов с основами работы с программным интерфейсом OpenGL. Для выполнения лабораторной работы студенту необходимо уметь ориентироваться в методах и типах используемых для растровой графики в OpenGL, уметь создавать и редактировать битовые и пиксельные маски, а также адаптировать их цветовое пространство.

Программный интерфейс OpenGL, кратко описанный в методических указаниях, может быть использован при создании моделей использующих конвейер трехмерной графики.

## **ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ**

Целью выполнения лабораторной работы является формирование практических навыков по работе с растровыми изображениями средствами OpenGL, а также простейшим преобразованиям цветовых и пространственных характеристик.

Основными задачами выполнения лабораторной работы являются: понимать принципы вывода растровых изображений, знать отличия битового образа от растрового и их характеристики, научиться использовать средства OpenGL для вывода растровых изображений, знать основные константы OpenGL, используемые при обработке растровых изображений, уметь создавать приложения OpenGL с использованием функций для работы с растровыми изображениями.

Результатами работы являются:

- Созданные средствами OpenGL образы
- Реализованные согласно варианту преобразования битовых и растровых образов
- Освоенные способы сокращенного описания растровых изображений
- Подготовленный отчет

## ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

До этого момента выполнение всех программ представляло результат преобразования и проектирования в двухмерное пространство трехмерных примитивов, которые растеризовались в буфере цвета. Однако OpenGL также поддерживает чтение непосредственно из буфера цвета и запись прямо в него. Это означает, что данные изображения можно считывать непосредственно из буфера цвета в буфер памяти, обрабатывая или записывая в файл. Это также означает, что можно считывать изображение из файла и помещать его непосредственно в буфер цвета. OpenGL позволяет не только считывать и записывать двухмерные изображения, но и поддерживает множество операций воспроизведения изображений, которые можно применять автоматически в процессе чтения и записи.

### Растровые изображения

Первые дисплеи компьютеров были монохромными (один цвет), обычно зелеными или желтыми, и все пиксели экрана могли находиться в одном из двух состояний: “включено” или “выключено”. В те дни компьютерная графика была очень простой, и данные об изображении представлялись *битовыми образами* (bitmap) — наборами нулей и единиц, обозначающими выключенные и включенные пиксели. В битовом образе каждый бит блока памяти соответствовал одному состоянию пикселя на экране. Битовые образы можно использовать в шрифтах и формах символов, масках (наложение фактуры на многоугольники) и даже двухцветных сглаженных изображениях. На *пиксельном образе* (pixelmap или pixmap) каждый пиксель имеет одну из 256 различных интенсивностей серого цвета. Термин *битовый образ* часто применяется к изображениям, содержащим полутоновые или полноцветные данные. В лабораторной работе под *битовым образом* понимается битовый (двоичный) образ, состоящий из значений “включено” и “выключено”, а *пиксельным* (или *растровым*) образом будем называть данные изображения, содержащие коды цвета или интенсивности пикселей.

### Упаковка пикселей

Из соображений производительности на многих аппаратных платформах каждая строка битового или пиксельного образа должна начинаться с выровненного по байтам адреса. Компиляторы автоматически помещают переменные и буферы по адресам, выровненным оптимальным для данной архитектуры образом. По умолчанию OpenGL предполагает 4-байтовое выравнивание, которое подходит для многих используемых систем. Битовый образ костра, использованный в предыдущем примере, был упакован плотно, но в данном случае это не представляет проблем, поскольку *сам* битовый образ имел 4-байтовое выравнивание. Напомним, что битовый образ имел ширину 32 бит (ровно 4 байт). Если бы использовали 34-битовый образ (всего на два бита больше), пришлось бы дополнять каждую строку 30 лишними битами неиспользуемого пространства и задействовать 64 бит. Хотя это может показаться ненужной тратой памяти, такое упорядочение позволяет процессору более эффективно захватывать блоки данных (например, такие, как строка битов битового образа).

Используя приведенные ниже функции, можно изменить схему хранения и извлечения пикселей битовых или растровых образов.

```
void glPixelStorei(GLenum pname, GLint param);  
void glPixelStoref(GLenum pname, GLfloat param);
```

Например, если требуется переключиться на плотную упаковку данных, вызывается следующая функция

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

Параметр `GL_UNPACK_ALIGNMENT` задает, как данные изображения будут распаковываться из буфера данных. Подобным образом можно использовать константу `GL_PACK_ALIGNMENT`, сообщая OpenGL, как паковать данные, считываемые из буфера цвета и помещаемые в заданный пользователем буфер памяти.

## Пиксельные образы

Гораздо больший интерес и больше сфер применения в современных полноцветных компьютерных системах представляют пиксельные образы. По смехе распределения памяти пиксельный образ похож на битовый, однако каждый пиксель может представляться несколькими битами памяти. Дополнительные биты позволяют записывать либо интенсивность (иногда называемую *сигналом яркости*), либо коды цветов-компонентов. Пиксельные образы рисуются в текущем

растровом положении так же, как и битовые, но используется при этом следующая функция:

```
void glDrawPixels(GLsizei width, GLsizei height, GLenum format, GLenum type, const void *pixels);
```

Первые два аргумента задают ширину и высоту изображения в пикселях. Третий аргумент задает формат данных изображения, затем следует тип данных и указатель на собственно данные. В отличие от `glBitmap` данная функция не обновляет растровое положение, кроме того она существенно гибче с точки зрения задания данных изображения. Каждый пиксель представляется одним или несколькими элементами данных, содержащихся в указателе `*pixel`. Цветовая схема этих элементов данных задается параметром `format` с помощью одной из констант, перечисленных в табл. 1. Параметр `type` интерпретирует данные, на которые указывает параметр `pixels`. Он сообщает OpenGL, какой тип данных используется в буфере для хранения компонентов цвета. Возможные значения аргументов рассматриваемой функции перечислены в табл. 2.

Таблица 1. Пиксельные форматы OpenGL

Константа	Описание
GL_RGB	Цвета в порядке “красный, зеленый, синий”
GL_RGBA	Цвета в порядке “красный, зеленый, синий, альфа”
GL_BGR/GL_BGR_EXT	Цвета в порядке “синий, зеленый, красный”
GL_BGRA/GL_BGRA_EXT	Цвета в порядке “синий, зеленый, красный, альфа”
GL_RED	Пиксели содержат только красный компонент
GL_GREEN	Пиксели содержат только зеленый компонент
GL_BLUE	Пиксели содержат только синий компонент



GL_ALPHA	Пиксели содержат только альфа-компонент
GL_LUMINANCE	Пиксели содержат только компонент яркости (интенсивности)
GL_LUMINANCE_ALPHA	Пиксели содержат яркости и альфа-компонент
GL_STENCIL_INDEX	Пиксели содержат только код трафарета
GL_DEPTH_COMPONENT	Пиксели содержат только код глубины

## Пиксельные форматы с упаковкой

Пиксельные форматы с упаковкой, перечисленные в табл. 2, впервые появились в OpenGL 1.2 как средство, позволяющее хранить данные об изображении в более сжатой форме, согласующейся с возможностями большого диапазона цветовоспроизводящего аппаратного обеспечения.



Рис.1. – Схема двух упакованных пиксельных форматов

Структура аппаратуры отображения информации позволяла экономить память или работать быстрее с небольшими наборами упакованных пиксельных данных. Эти пиксельные форматы

все еще используются некоторым аппаратным обеспечением и могут пригодиться будущим аппаратным платформам.

Названные пиксельные форматы сжимают данные о цвете до минимального числа битов, причем число битов на цветовой канал указывается в соответствующей константе. Например, формат `GL_UNSIGNED_BYTE_3_3_2` записывает три бита первого компонента, три бита второго и два бита третьего компонента.

Таблица 2. Типы пиксельных данных

Константа	Описание
<code>GL_UNSIGNED_BYTE</code>	Все компоненты цвета является 8-битовыми целыми числами без знака
<code>GL_BYTE</code>	8-битовые числа со знаком
<code>GL_BITMAP</code>	Отдельные биты без данных о цвете, то же, что <code>glBitmap</code>
<code>GL_UNSIGNED_SHORT</code>	16-битовые целые числа без знака
<code>GL_SHORT</code>	16-битовые целые числа со знаком
<code>GL_UNSIGNED_INT</code>	32-битовые целые числа без знака
<code>GL_INT</code>	32-битовые целые числа со знаком
<code>GL_FLOAT</code>	Величины с плавающей запятой обычной точности
<code>GL_UNSIGNED_BYTE_3_2_2</code>	Упакованные RGB-коды
<code>GL_UNSIGNED_SHORT_4_4_4_4</code>	Упакованные RGBA-коды

Помните, что конкретные компоненты (красный, зеленый, синий и альфа) упорядочены согласно параметру `format` функции `glDrawPixels`. Компоненты упорядочиваются от старших битов (most significant bit — MSB) до младших (least

significant bit — LSB). `GL_UNSIGNED_BYTE_2_3_3_REV` обрабатывает этот порядок и помещает последний компонент в два старших бита и т.д. На рис. 1 графически показана побитовая схема данных двух упорядочений.

## Перемещение пикселей

Запись данных о пикселях в буфер цвета может оказаться очень полезной сама по себе, но можно считывать данные из буфера цвета и даже копировать их из одной части буфера цвета в другую. Функция, считывающая данные о пикселях, действует подобно `glDrawPixels` (только наоборот)

```
void glReadPixels(GLint x, GLint y, GLsizei
width, GLsizei height, GLenum format, GLenum type, const
void *pixels);
```

В координатах окна задаются  $x$  и  $y$  — координаты левого нижнего угла прямоугольника, который следует считать, после чего указывается ширина и высота прямоугольника в пикселях. Параметры `format` и `type` представляют формат и тип данных. Если буфер цвета хранит данные не так, как запросили, OpenGL выполнит все необходимые преобразования. Указатель на данные изображения `*pixels` должен быть действительным и предусматривать достаточно места для хранения данных об изображении после преобразования.

Копирование пикселей из одной части буфера цвета в другую также является довольно простой операцией, при выполнении которой не требуется выделять временной памяти для хранения информации. Вначале с помощью `glRasterPos` или `glWindowPos` задается растровое положение — целевой угол (помните, — левый нижний угол), куда необходимо скопировать данные изображения. Затем для выполнения операции копирования используется следующая функция:

```
void glCopyPixels(GLint x, GLint y, GLsizei
width, GLsizei height, GLenum type);
```

Параметры  $x$  и  $y$  задают левый нижний угол копируемого прямоугольника, после чего указывается его ширина и высота в пикселях. Параметр `type` должен иметь значение `GL_COLOR`, соответствующее копированию данных о цвете. Здесь также можно использовать

константы `GL_DEPTH` и `GL_STENCIL`, тогда копирование будет выполняться в буфер глубины.

По умолчанию все описанные операции с пикселями действуют в заднем буфере в контексте визуализации с двойной буферизацией и в переднем буфере в контексте визуализации с простой буферизацией. Источник или цель данных операций можно изменить, используя следующие функции:

```
void glDrawBuffer(GLenum mode);  
void glReadBuffer(GLenum mode);
```

Функция `glDrawBuffer` определяет, какие пиксели будут рисоваться при выполнении операции `glDrawPixels` или `glCopyPixels`. Здесь можно использовать любую из допустимых констант буфера: `GL_NONE`, `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT` и т.д.

Функция `glReadBuffer` принимает те же константы и задает целевой буфер цвета для операции чтения, выполняемой функцией `glReadPixels` или `glCopyPixels`.

## Передача пикселей

Помимо масштабирования пикселей OpenGL поддерживает ряд простых математических операций, которые можно выполнить с данными, передаваемыми из буфера цвета или в него. Эти режимы передачи пикселей устанавливаются с помощью одной из следующих функций, а параметры передачи пикселей перечислены в табл. 5.3.

```
void glPixelTransferi(GLenum pname, GLint param);  
void glPixelTransferf(GLenum pname, GLfloat param);
```

Параметры масштабирования и смещения позволяют масштабировать и смещать отдельные цветовые каналы. Масштабный коэффициент умножается на код компонента, а код смещения прибавляется к коду компонента. Определенные таким образом операции масштабирования и смещения широко распространены в компьютерной графике, связанной с настройкой значений цветовых каналов. Уравнение, согласно которому производится расчет, записывается достаточно просто:

новое значение = (старое значение \* масштабный коэффициент) + смещение

Таблица 3. Параметры передачи пикселей

Константа	Тип	Значение по умолчанию
<b>GL_MAP_COLOR</b>	<b>GLboolean</b>	<b>GL_FALSE</b>
<b>GL_MAP_STENCIL</b>	<b>GLboolean</b>	<b>GL_FALSE</b>
<b>GL_RED_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_GREEN_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_BLUE_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_ALPHA_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_DEPTH_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_RED_BIAS</b>	<b>GLfloat</b>	<b>0.0</b>
<b>GL_GREEN_BIAS</b>	<b>GLfloat</b>	<b>0.0</b>
<b>GL_BLUE_BIAS</b>	<b>GLfloat</b>	<b>0.0</b>
<b>GL_ALPHA_BIAS</b>	<b>GLfloat</b>	<b>0.0</b>
<b>GL_DEPTH_BIAS</b>	<b>GLfloat</b>	<b>0.0</b>
<b>GL_POST_CONVOLUTION_RED_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_POST_CONVOLUTION_GREEN_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_POST_CONVOLUTION_BLUE_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>
<b>GL_POST_CONVOLUTION_ALPHA_SCALE</b>	<b>GLfloat</b>	<b>1.0</b>

По умолчанию масштабные коэффициенты равны 1.0, а коды смещения — 0.0. По сути, такие величины не влияют на коды компонентов. В качестве примера предположим, что требуется отобразить

только красную составляющую изображения. Для этого масштабные коэффициенты синего и зеленого каналов устанавливаются равными 0.0, а после изображения рисунка возвращаются к исходному значению 1.0.

```
glPixelTransferf(GL_GREEN_SCALE, 0.0f);  
glPixelTransferf(GL_BLUE_SCALE, 0.0f);
```

Для иллюстрации данной концепции в программу включено меню с позициями Just Red, Just Green и Just Blue. Выбор любой из этих позиций отключает все каналы, кроме одного, в результате на изображении присутствуют только коды красного, зеленого или синего компонентов.

```
case 4:  
    // Только красный  
    glPixelTransferf(GL_RED_SCALE, 1.0f);  
glPixelTransferf(GL_GREEN_SCALE, 0.0f);  
glPixelTransferf(GL_BLUE_SCALE, 0.0f); break;  
case 5:  
    // Только зеленый  
    glPixelTransferf(GL_RED_SCALE, 0.0f);  
    glPixelTransferf(GL_GREEN_SCALE, 1.0f);  
glPixelTransferf(GL_BLUE_SCALE, 0.0f); break;  
case 6:  
    // Только синий  
    glPixelTransferf (GL_RED_SCALE, 0.0f) ;  
glPixelTransferf(GL_GREEN_SCALE, 0.0f);  
glPixelTransferf(GL_BLUE_SCALE, 1.0f); break;
```

После рисования с помощью переноса пикселей масштабные коэффициенты возвращаются к исходному значению - 1.0

```
glPixelTransferf(GL_RED_SCALE, 1.0f);  
glPixelTransferf(GL_GREEN_SCALE, 1.0f);  
glPixelTransferf(GL_BLUE_SCALE, 1.0f);
```

Такого же результата можно добиться с помощью параметров масштабирования и смещения, но тогда придется ожидать завершения операций свертки или действий с матрицей цветов. Эти операции доступны в подмножестве построения изображений, которое рассмотрим ниже.

Более интересным примером передачи пикселей является черно-белое отображение цветного изображения. Увидеть такое изображе-

ние в программе можно, выбрав из меню опцию Black and White. Итак, вначале в буфере цвета рисуется полноцветное изображение

```
glDrawPixels(lWidth, iHeight, eFormat,  
GL_UNSIGNED_BYTE, plmage);
```

Затем выделяется буфер, настолько большой, чтобы вместить коды яркости всех пикселей

```
pModifiedBytes = (GLbyte *)malloc(lWidth *  
iHeight);
```

Помните, что яркостному изображению соответствует всего один цветовой канал, поэтому для его хранения выделяется 1 байт (8 бит) на пиксель. При вызове `glReadPixels` OpenGL автоматически преобразовывает изображение в буфере цвета в коды яркости, но требует, чтобы данные имели формат `GL_LUMINANCE`.

```
glReadPixels(0,0,iWidth, iHeight, GL_LUMINANCE,  
GL_UNSIGNED_BYTE, pModifiedBytes);
```

Затем яркостное изображение снова можно записать в буфер цвета и увидеть преобразованное черно-белое изображение

```
glDrawPixels(lWidth, iHeight, GL_LUMINANCE,  
GL_UNSIGNED_BYTE, pModifiedBytes);
```

Использование описанного подхода кажется привлекательным, и он почти работает. Проблема заключается в том, что, когда OpenGL преобразовывает цветное изображение в яркостное, компоненты цветных каналов просто суммируются. Если три цветовых канала при суммировании дадут значение больше 1, оно будет просто приравнено к 1. Таким образом создается эффект перенасыщенности многих областей изображения.

Чтобы решить возникшую проблему, режим передачи пикселей нужно установить так, чтобы коды цвета соответствующим образом масштабировались при переходе из пространства цветов к яркости. Согласно стандарт Национального комитета по телевизионным стандартам (National Television Standards Committee - NTSC), преобразование из пространства цветов RGB к черно-белому (полутоновому) изображению происходит следующим образом:

$$\text{яркость} = (0.3 * \text{красный}) + (0.59 * \text{зеленый}) + (0.11 * \text{синий})$$

Такое преобразование можно легко задать в OpenGL, вызвав необходимые функции непосредственно перед выполнением `glReadPixels`.

```
// Масштабируем цвета согласно стандарту NTSC
```

```
glPixelTransferf(GL_RED_SCALE, 0.3f); glPixel-
Transferf(GL_GREEN_SCALE, 0.59f);
glPixelTransferf(GL_BLUE_SCALE, 0.11f);
```

После считывания пикселей режим передачи пикселей возвращается к нормальному.

```
// Масштабирование цвета возвращается в норму
glPixelTransferf(GL_RED_SCALE, 1.0f); glPixel-
Transferf(GL_GREEN_SCALE, 1.0f); glPixelTrans-
ferf(GL_BLUE_SCALE, 1.0f);
```

Теперь результатом выполнения программы является приятное полутоновое изображение.

## Отображение пикселей

Помимо операций масштабирования и смещения в число операций переноса пикселей входит отображение цвета. *Картой цветов* называется справочная таблица, используемая для преобразования одного кода цвета (индекса-указателя в таблицу) в другой код цвета (действительный код цвета, записанный в позиции, указываемой индексом) Отображение цвета применяется во многих областях, например, цветокоррекции, гамма-коррекции или преобразованиях между различными представлениями цветов. Исследуя программу и выбрав опцию Invert Colors, можно обнаружить интересный пример. В этом случае устанавливается карта цветов, в процессе переноса пикселей обращающая все коды цвета. Это означает, что все три канала отображаются из диапазона 0.0-1.0 в диапазон 1.0-0.0. В результате получается изображение, выглядящее, как негатив фотографии

Таблица 4. Параметры карты пикселей

Имя карты
GL_PIXEL_MAP_R_TO_R
GL_PIXEL_MAP_G_TO_G
GL_PIXEL_MAP_B_TO_B
GL_PIXEL_MAP_A_TO_A

Чтобы активизировать отображение пикселей, вызывается функция `glPixelTransfer` с параметром `GL_MAP_COLOR`, имеющим значение `GL_TRUE`.



```
glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
```

Чтобы установить карту пикселей, нужно вызвать другую функцию — `glPixelMap` и предоставить карту в одном из трех форматов.

```
glPixelMapuiv(GLenum map, GLint mapsize, GLuint
*values) ; glPixelMapusv(GLenum map, GLint mapsize,
GLushort *values) ; glPixelMapfv(GLenum map, GLint
mapsize, GLfloat *values) ;
```

Допустимы значения функции перечислены в табл. 4.

В этом примере устанавливается карта из 256 величин типа `float` и заполняется карта промежуточными значениями от 1.0 до 0.0.

```
GLfloat invertMap[256];
...
invertMap[0] = 1.0f;
for(l = 1; l < 256; i++)
invertMap[l] = 1.0f - (1.0f/255.Of *
(GLfloat)i);
```

Затем устанавливаем красную, зеленую и синюю карты этой обращенной карты и включаем отображение цвета

```
glPixelMapfv(GL_PIXEL_MAP_R_TO_R, 255,
invertMap); glPixelMapfv(GL_PIXEL_MAP_G_TO_G, 255,
invertMap); glPixelMapfv(GL_PIXEL_MAP_B_TO_B, 255,
invertMap); glPixelTransferi(GL_MAP_COLOR,
GL_TRUE);
```

При вызове `glDrawPixels` цвета-компоненты повторно отображаются с помощью таблицы обращения, по сути, создавая цветной негатив, показанный на рис. 2

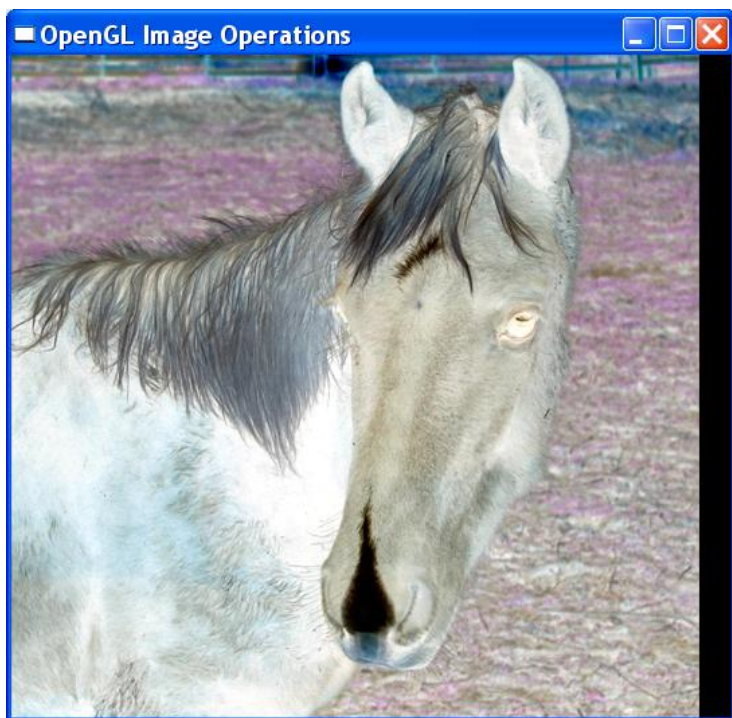


Рис.2. – Использование карты цветов для создания цветного негати-  
тива

## ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

### Пример битового образа

В листинге 1 приведена программа, использующая те же битовые данные, что применялись в предыдущих лабораторных работах для наложения фактуры на многоугольник (маленький костер, представленный битовым “узором” 32 x 32). Помните, что битовые образы строятся снизу вверх, т.е. первая строка данных представляет нижнюю строку битового образа. Программа создает окно 512x512 и заполняет его изображением костра из 16 строк и 16 столбцов. Результат выполнения программы показан на рис. 3. Обратите внимание на то, что функция **ChangeSize** задает ортографическую проекцию, сохраняя высоту и ширину окна в пикселях.

Листинг 1 – Программа вывода множественного изображения

```
#include "glew.h"
#include "glut.h"

// Растровый образ костра
GLubyte fire[128] = { 0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0x00,
                      0x00, 0x00, 0x00, 0xc0,
                      0x00, 0x00, 0x01, 0xf0,
                      0x00, 0x00, 0x07, 0xf0,
                      0x0f, 0x00, 0x1f, 0xe0,
                      0x1f, 0x80, 0x1f, 0xc0,
                      0x0f, 0xc0, 0x3f, 0x80,
                      0x07, 0xe0, 0x7e, 0x00,
                      0x03, 0xf0, 0xff, 0x80,
                      0x03, 0xf5, 0xff, 0xe0,
                      0x07, 0xfd, 0xff, 0xf8,
                      0x1f, 0xfc, 0xff, 0xe8,
                      0xff, 0xe3, 0xbf, 0x70,
                      0xde, 0x80, 0xb7, 0x00,
                      0x71, 0x10, 0x4a, 0x80,
                      0x03, 0x10, 0x4e, 0x40,
```

```

0x02, 0x88, 0x8c, 0x20,
0x05, 0x05, 0x04, 0x40,
0x02, 0x82, 0x14, 0x40,
0x02, 0x40, 0x10, 0x80,
0x02, 0x64, 0x1a, 0x80,
0x00, 0x92, 0x29, 0x00,
0x00, 0xb0, 0x48, 0x00,
0x00, 0xc8, 0x90, 0x00,
0x00, 0x85, 0x10, 0x00,
0x00, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x00 };

////////////////////////////////////
////////////////////////////////////
// Функция выполняет всю необходимую инициализацию в
контексте визуализации
void SetupRC()
{
    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}
////////////////////////////////////
////////////////////////////////////
// Устанавливается система координат, согласованная с
координатами
void ChangeSize(int w, int h)
{
    // Предотвращает деление на нуль, когда окно слишком
маленькое
    // (нельзя сделать окно нулевой ширины).
    if(h == 0)
        h = 1;

    glViewport(0, 0, w, h);

    // Система координат обновляется перед модифика-
цией
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Псевдокоординаты окна

```

```

        gluOrtho2D(0.0, (GLfloat) w, 0.0f, (GLfloat)
h);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

////////////////////////////////////
////////////////////////////////////
// Вызывается для рисования сцены
void RenderScene(void)
{
    int x, y;

    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT);

    // Устанавливается белый цвет
    glColor3f(1.0f, 1.0f, 1.0f);

    // Цикл из 16 строк и столбцов
    for(y = 0; y < 16; y++)
    {
        // Устанавливается растровое положение
данного "квадрата"          glRasterPos2i(0, y *
32);
        for(x = 0; x < 16; x++)
            // Рисуются битовый образ "костра",
меняется растровое положение
            glBitmap(32, 32, 0.0, 0.0, 32.0,
0.0, fire);
    }

    // Переключает буферы
    glutSwapBuffers();
}

////////////////////////////////////
////////////////////////////////////
// Точка входа основной программы
int main(int argc, char* argv[])

```

```

{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
glutInitWindowSize(512, 512);
glutCreateWindow("OpenGL Bitmaps");
glutReshapeFunc(ChangeSize);
glutDisplayFunc(RenderScene);

SetupRC();
glutMainLoop();

return 0;
}

```

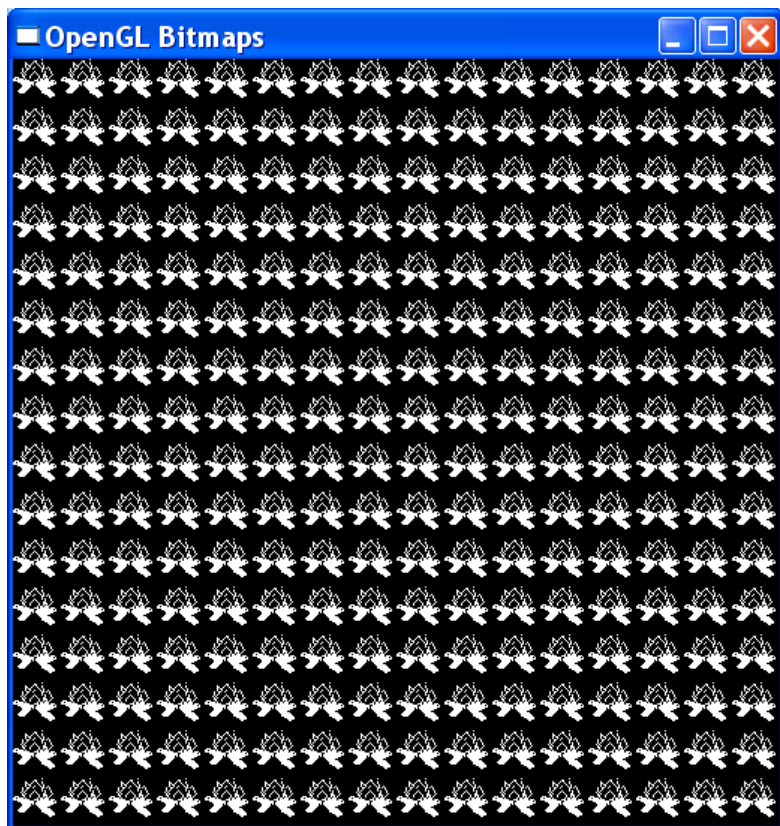


Рис.3. – Битовый образ костра

## Установка растрового положения

Суть программы из листинга 1. заключена в функции **RenderScene**, где вложенные циклы рисуют 16 строк и 16 столбцов растрового образа костра.

```
// Цикл из 16 строк и столбцов
for(y =0; y < 16; y++)
{
    // Устанавливается растровое положение "квадрата"
    glRasterPos2i(0, y * 32);
    for(x = 0; x < 16; x++)
    // Рисуются битовый образ "костра",
    // меняется растровое положение
    glBitmap(32, 32, 0.0, 0.0, 32.0, 0.0, fire);
}
```

Первый цикл (переменная *y*) проходит по строкам с 0 по 16. Вызов следующей процедуры устанавливает растровое положение точки, в которой необходимо изобразить битовый образ.

```
glRasterPos2i(0, y * 32);
```

Растровое положение интерпретируется как вызов функции **glVertex**, т.е. с помощью текущих матриц наблюдения модели и проекции так же преобразовываются координаты.

В этом примере проекция OpenGL соответствовала размерам окна, поэтому для размещения битовых образов можем использовать координаты окна. Тем не менее данная техника не всегда удобна, так что OpenGL предлагает альтернативную функцию, позволяющую устанавливать растровое положение в координатах окна безотносительно к текущей матрице преобразования или проекции.

```
void glWindowPos2i(GLint x, GLint y);
```

Функция **glWindowPos** может иметь два или три аргумента и принимает величины типа *integer*, *float*, *double* и *short* (как и функция **glVertex**).

Цвет растрового образа задается путем вызова функции **glRasterPos** либо **glWindowPos**. Это означает, что текущий цвет, установленный ранее с помощью **glColor**, ограничивается последующими растровыми операциями. Вызов **glColor** после установки растрового положения никак не повлияет на цвет битового образа.

## Рисование битового образа

Следующая команда рисует битовый образ в буфере цвета  
`glBitmap(32, 32, 0.0, 0.0, 32.0, 0.0, fire);`

Функция `glBitmap` копирует указанный битовый образ в буфер цвета в текущем растровом положении и (необязательно) меняет растровое положение — все в одной операции. Эта функция имеет следующий синтаксис

```
void glBitmap(GLsize width, GLsize height,
GLfloat xorg, GLfloat yorg, GLfloat xmove, GLfloat
ymove, GLubyte *bitmap);
```

Первые два параметра `width` и `height` задают ширину и высоту битового образа (в битах). Следующие два параметра `xorg` и `yorg` (величины типа `float`) задают начало битового образа. Чтобы началом считался левый нижний угол образа, обоим аргументам присваивается значение `0.0`. Следующие аргументы, `xmove` и `ymove`, задают смещение в пикселях растрового положения в направлениях `x` и `y` после визуализации растрового образа. Обратите внимание на то, что эти четыре параметра представлены величинами с плавающей запятой. Последний аргумент `bitmap` — просто указатель на данные битового образа.

## Пример работы с растровым образом

Пришло время использовать полученные знания о пикселях для создания более яркого и реалистичного изображения костра. На рис.4. показан результат выполнения программы, которая загружает изображение `fire.tga` и использует функцию `glDrawPixels` для помещения этого рисунка непосредственно в буфер цвета. Программа почти идентичная предыдущей, только данные об изображении вначале считываются из файла `targa` (обратите внимание на расширение `.tga`) с помощью собственной функции `glLoadTGA`, а затем рисуются с помощью `glDrawPixels`, а не `glBitmap`, как было раньше. Функция, загружающая и отображающая файл на экране, приведена в листинге 2.





Рис.4. –Изображение костра, загруженное из файла

Листинг 2 – Функция RenderScene, загружающая и отображающая файл изображения

```
void RenderScene(void)
{
    GLubyte *pImage = NULL;
    GLint iWidth, iHeight, iComponents;
    GLenum eFormat;

    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT);

    // Информация в файле targa выравнена по одному
    // байту
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

```

//Загружает файл TGA, дает информацию по ширине, вы-
соте
//и компонентам/формату
    pImage = (GLubyte*)glTLoadTGA("fire.tga",
&iWidth, &iHeight, &iComponents, &eFormat);

    // Для задания растрового положения применяются
координаты окна
    glRasterPos2i(0, 0);

    // Рисуется пиксельный образ
    if(pImage != NULL)
        glDrawPixels(iWidth, iHeight, eFormat,
GL_UNSIGNED_BYTE, pImage);

    // Данные изображения уже не нужны
    free(pImage);

    // Переключает буферы
    glutSwapBuffers();
}

```

Рассмотрим вызов функции, считывающей файл targa

```

// Загружает файл TGA, дает информацию по ширине,
высоте
// и компонентам/формату
pImage = glTLoadTGA("fire.tga", SiWidth, SiHeight,
SiComponents, seFormat);

```

Первым аргументом этой функции является имя загружаемого файла targa (если необходимо — с путем). Формат targa является широко поддерживаемым и распространенным форматом изображений. В отличие от файлов JPEG файлы targa (обычно) хранят изображение в несжатой форме. Функция glTLoadTGA открывает файл, а затем считывает и разбирает его заголовок, определяя ширину, высоту и формат данных файла. Число компонентов может быть равно одному, трем или четырем и представлять RGB- или RGBA- изображение, соответственно. Последний параметр — это указатель на величину типа GLenum, получающую формат изображения OpenGL, соответствующий файлу. При успешном вызове функции он возвращает только что присвоенный указатель на данные изображения, считанные непосредственно из файла. Если файл не найден или произошла другая

ошибка, функция возвращает NULL. Полное описание функции `gltLoadTGA` приведено в листинге 3.

Листинг 3 – Функция `gltLoadTGA`, загружающая файлы Targa для использования OpenGL

```
// Определение Targa заголовка.
#pragma pack(1)
typedef struct
{
    GLbyte identsize;    //Размер поля ID, который
    соответствует заголовку (0)
    GLbyte colorMapType;    // 0 = None, 1 =
    paletted
    GLbyte imageType;    // 0 = none, 1 =
    indexed, 2 = rgb, 3 = grey, +8=rle
    unsigned short    colorMapStart;    //
    Первая позици цветной карты
    unsigned short    colorMapLength;    // Ко-
    личество цветов
    unsigned char    colorMapBits;    // Коли-
    чество бит
    unsigned short    xstart;    // на-
    чало координаты X
    unsigned short    ystart;    // на-
    чало координаты Y
    unsigned short    width;    // ши-
    рина в пикселях
    unsigned short    height;    // вы-
    сота в пикселях
    GLbyte bits;    // количество бит
    на пикселе (8 16, 24, 32)
    GLbyte descriptor;    // Дескриптор
    изображения
} TGAHEADER;
#pragma pack(8)

////////////////////////////////////
////////////////////////////////////
```

```

// Распределяет память и загружает биты файла targa.
Возвращает
// указатель на новые буфер, высоту и ширину тексту-
ры, формат
// данных OpenGL. Вызывает free() для освобождения
буфера
// после завершения. Работает только с простыми уни-
фицированными
// файлами targas с 8-, 24- или 32-битовым цветом,
без палитр,
// без группового кодирования
GLbyte *gltLoadTGA(const char *szFileName, GLint
*iWidth, GLint *iHeight, GLint *iComponents, GLenum
*eFormat)
{
    FILE *pFile;                // Указатель файла
    TGAHEADER tgaHeader;        // Заголовок файла
TGA
    unsigned long lImageSize;    // Размер
изображения в байтах
    short sDepth;               // Размер пикселя
;
    GLbyte *pBits = NULL;       // Указатель на
биты

// Значения по умолчанию/значения при сбое
    *iWidth = 0;
    *iHeight = 0;
    *eFormat = GL_BGR_EXT;
    *iComponents = GL_RGB8;

    // Пытаемся открыть файл
    pFile = fopen(szFileName, "rb");
    if(pFile == NULL)
        return NULL;

    // Считываем заголовок (двоичный)
    fread(&tgaHeader, 18/* sizeof(TGAHEADER)*/, 1,
pFile);

// Обращение байтов при переходе между обратным и
прямым

```

```

// порядком битов
#ifdef __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);
    BYTE_SWAP(tgaHeader.xstart);
    BYTE_SWAP(tgaHeader.ystart);
    BYTE_SWAP(tgaHeader.width);
    BYTE_SWAP(tgaHeader.height);
#endif

// Получаем ширину, высоту и глубину текстуры
*iWidth = tgaHeader.width;
*iHeight = tgaHeader.height;
sDepth = tgaHeader.bits / 8;

// Проверки приемлемости. Очень просто: я понимаю
только
// 8-, 24- или 32-битовые файлы targa
if(tgaHeader.bits != 8 && tgaHeader.bits != 24 &&
tgaHeader.bits != 32)
    return NULL;

// Расчет размера буфера изображения
lImageSize = tgaHeader.width * tgaHeader.height *
sDepth;

// Распределение памяти и проверка успешности
pBits = (GLbyte*)malloc(lImageSize *
sizeof(GLbyte));
if(pBits == NULL)
    return NULL;

// Считывание битов
// Проверка на наличие ошибок чтения. Здесь должны //
отлавливаться групповое кодирование или другие
// форматы, которые не нужно распознавать
if(fread(pBits, lImageSize, 1, pFile) != 1)
{
    free(pBits);
    return NULL;
}

```

```

// Устанавливается формат, ожидаемый OpenGL
switch(sDepth)
{
case 3:      // Наиболее вероятный случай
    *eFormat = GL_BGR_EXT;
    *iComponents = GL_RGB8;
    break;
case 4:
    *eFormat = GL_BGRA_EXT;
    *iComponents = GL_RGBA8;
    break;
case 1:
    *eFormat = GL_LUMINANCE;
    *iComponents = GL_LUMINANCE8;
    break;
};

// Работа с файлом закончена
fclose(pFile);

// Возвращает указатель на данные изображения
return pBits;
}

```

Число компонентов устанавливается равным не целым числам 1, 3 или 4, а `GL_LUMINANCE8`, `GL_RGB8` и `GL_RGBA8`. Когда OpenGL манипулирует данными изображения, он распознает специальные константы как запрос на внутреннее поддержание полной точности воспроизведения изображения. Например, из соображений производительности некоторые реализации OpenGL могут внутренне перевыбирать изображение с 24-битовым цветом до 16-битового цвета. Особенно такой подход распространен при загрузке текстуры во многих реализациях, где цветовая разрешающая способность вывода на дисплей равна всего 16 бит, а загружается изображение с большей насыщенностью цвета. Константы представляют собой запросы к OpenGL, выдаваемые с целью хранения и использования данных изображения, предоставленных с полной насыщенностью — 8 бит на канал.

## Запись пикселей

Теперь достаточно известно о том, как перемещать пиксели, чтобы записать еще одну полезную функцию. Дополнением к функции загрузки файлов targa `glLoadTGA` является `glWriteTGA`. Эта функция считывает данные о цвете из переднего буфера цвета и записывает их в файл изображения в формате targa. В следующем разделе данная функция используется при работе с пиксельными операциями OpenGL. Полная информация о функции `glWriteTGA` приводится в листинге 4.

Листинг 4 – Функция `glWriteTGA`, записывающая изображение на экране в файл Targa

```
////////////////////////////////////  
////////////////////////////////////  
// Записывается текущее поле просмотра как файл targa  
// Перед вызовом этой функции не забудьте вызвать  
SwapBuffers  
// для использование двойной буферизации или glFinish  
для простой  
// буферизации. Возвращает 0, если происходит ошибка,  
и 1,  
// если все хорошо  
GLint glWriteTGA(const char *szFileName)  
{  
    FILE *pFile;                // Указатель файла  
    TGAHEADER tgaHeader;        // Заголовок файла  
TGA  
    unsigned long lImageSize;    // размер изображения  
в байтах  
    GLbyte *pBits = NULL;       // Указатель на биты  
    GLint iViewport[4];         // Размер поля про-  
смотра в пикселях  
    GLenum lastBuffer;          // Память для хране-  
ния текущих настроек буфера чтения  
    // Получает размеры поля просмотра  
    glGetIntegerv(GL_VIEWPORT, iViewport);  
  
    // Насколько большим будет изображение (файлы  
targa плотно упакованы)
```

```

    lImageSize = iViewport[2] * 3 * iViewport[3];

    // Распределяет блок. Если это не работает, воз-
    вращаемся домой
    pBits = (GLbyte *)malloc(lImageSize);
    if(pBits == NULL)
        return 0;

    // Считывает биты из буфера цвета
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
    glPixelStorei(GL_PACK_ROW_LENGTH, 0);
    glPixelStorei(GL_PACK_SKIP_ROWS, 0);
    glPixelStorei(GL_PACK_SKIP_PIXELS, 0);

    // Получает текущие установки буфера чтения и за-
    писывает их.
    // Переключается на передний буфер и выполняет опера-
    цию чтения.
    //В конце концов восстанавливает состояние буфера
    чтения
    glGetIntegerv(GL_READ_BUFFER,
    (GLint*)&lastBuffer);
    glReadBuffer(GL_FRONT);
    glReadPixels(0, 0, iViewport[2], iViewport[3],
    GL_BGR_EXT, GL_UNSIGNED_BYTE, pBits);
    glReadBuffer(lastBuffer);

    // Инициализирует заголовок файла Targa
    tgaHeader.identsize = 0;
    tgaHeader.colorMapType = 0;
    tgaHeader.imageType = 2;
    tgaHeader.colorMapStart = 0;
    tgaHeader.colorMapLength = 0;
    tgaHeader.colorMapBits = 0;
    tgaHeader.xstart = 0;
    tgaHeader.ystart = 0;
    tgaHeader.width = iViewport[2];
    tgaHeader.height = iViewport[3];
    tgaHeader.bits = 24;
    tgaHeader.descriptor = 0;

```



```

        // Обращение байтов при переходе между обратным и
        // прямым II порядком битов

#ifdef __APPLE__
    BYTE_SWAP(tgaHeader.colorMapStart);
    BYTE_SWAP(tgaHeader.colorMapLength);
    BYTE_SWAP(tgaHeader.xstart);
    BYTE_SWAP(tgaHeader.ystart);
    BYTE_SWAP(tgaHeader.width);
    BYTE_SWAP(tgaHeader.height);
#endif

    // Пытается открыть файл
    pFile = fopen(szFileName, "wb");
    if(pFile == NULL)
    {
        free(pBits);    // Освобождает буфер и воз-
        вращает ошибку
        return 0;
    }

    // Записывает заголовок
    fwrite(&tgaHeader, sizeof(TGAHEADER), 1, pFile);

    // Записывает заголовок
    fwrite(pBits, lImageSize, 1, pFile);

    // Освобождает временный буфер и закрывает файл
    free(pBits);
    fclose(pFile);

    // Успех!
    return 1;
}

```

## Обработка изображений

В данном разделе обсуждается поддержка OpenGL увеличения и уменьшения изображения, переворота изображений и выполнения нескольких специальных операций в процессе передачи данных о пикселях в буфер цвета и из него. Вместо того чтобы создавать отдельную

программу для демонстрации каждого обсуждаемого специального эффекта было разработано одно приложение. Эта программа отображает простое цветное изображение, загруженное из файла targa. Щелчок правой кнопкой мыши соотнесен с системой меню GLUT, позволяющей выбирать один из восьми режимов рисования или сохранять модифицированное изображение на диске в файле screenshot1.tga. В листинге 5 программа приведена целиком. В последующих разделах она разбирается по отдельным фрагментам и подробно описывается.

### Листинг 5 – Исходный код программы

```
#include "glew.h"
#include "glut.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

////////////////////////////////////
////////////////////////////////////
// Глобальные переменные модуля для записи исходных
// данных изображения static GLubyte *pImage = NULL;
static GLint iWidth, iHeight, iComponents;
static GLenum eFormat;

// Глобальные переменные для хранения режима рисова-
// ния
static GLint      iRenderMode = 1;

// Определение заголовка targa
#pragma pack(1)
typedef struct
{
    GLbyte identsize;      //Размер поля ID, который
    соответствует заголовку (0)
    GLbyte colorMapType;   // 0 = None, 1 =
    paletted
    GLbyte imageType;      // 0 = none, 1 =
    indexed, 2 = rgb, 3 = grey, +8=rle
    unsigned short colorMapStart; //
    Первая позиция цветной карты
}
```

```

        unsigned short    colorMapLength;           // Ко-
личество цветов
        unsigned char     colorMapBits;            // Коли-
чество бит
        unsigned short    xstart;                  // на-
чало координаты X
        unsigned short    ystart;                  // на-
чало координаты Y
        unsigned short    width;                   // ши-
рина в пикселях
        unsigned short    height;                  // вы-
сота в пикселях
        GLbyte bits;                                // количество бит
на пикселе(8 16, 24, 32)
        GLbyte descriptor;                          // Дескриптор
изображения
    } TGAHEADER;

#pragma pack(8)

////////////////////////////////////
////////////////////////////////////
// Эта функция выполняет необходимую инициализацию в
контексте
// визуализации.
void SetupRC(void)
{
    // Черный фон
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

    // Загружаем изображение лошади
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    pImage = (GLubyte*)glTLoadTGA("horse.tga",
&iWidth, &iHeight, &iComponents, &eFormat);
}

void ShutdownRC(void)
{
    // Освобождаем исходные данные изображения
    free(pImage);
}

```

```

////////////////////////////////////
////////////////////////////////////
// Должным образом обновляем флаги в ответ на выбор
// позиции из меню

void ProcessMenu(int value)
{
    if(value == 0)
        // Записываем изображение
        gltWriteTGA("ScreenShot1.tga");
    else
        // Меняем индекс режима визуализации на ин-
        декс,
        // соответствующий позиции меню

        iRenderMode = value;

    // Активизируем перерисовывание изображения
    glutPostRedisplay();
}

////////////////////////////////////
////////////////////////////////////
// Вызывается для рисования сцены
void RenderScene(void)
{
    GLint iViewport[4];
    GLbyte *pModifiedBytes = NULL;
    GLfloat invertMap[256];
    GLint i;

    // Очищаем окно текущим цветом очистки
    glClear(GL_COLOR_BUFFER_BIT);

    // Текущее растровое положение всегда соответствует
    // левому
    // нижнему углу окна
    glRasterPos2i(0, 0);

```

```

//В зависимости от индекса режима визуализации выпол-
няются
// необходимые операции с изображением
switch(iRenderMode)
{
    case 2:      // Переворачиваем пиксели
        glPixelZoom(-1.0f, -1.0f);
        glRasterPos2i(iWidth, iHeight);
        break;

    case 3:      // Увеличиваем пиксели для запол-
        нения окна
        glGetIntegerv(GL_VIEWPORT, iViewport);
        glPixelZoom((GLfloat) iViewport[2] /
(GLfloat)iWidth, (GLfloat) iViewport[3] /
(GLfloat)iHeight);
        break;

    case 4:      // Только    красный
        glPixelTransferf(GL_RED_SCALE, 1.0f);
        glPixelTransferf(GL_GREEN_SCALE, 0.0f);
        glPixelTransferf(GL_BLUE_SCALE, 0.0f);
        break;

    case 5:      // Только    зеленый
        glPixelTransferf(GL_RED_SCALE, 0.0f);
        glPixelTransferf(GL_GREEN_SCALE, 1.0f);
        glPixelTransferf(GL_BLUE_SCALE, 0.0f);
        break;

    case 6:      // Только    синий
        glPixelTransferf(GL_RED_SCALE, 0.0f);
        glPixelTransferf(GL_GREEN_SCALE, 0.0f);
        glPixelTransferf(GL_BLUE_SCALE, 1.0f);
        break;

    case 7:      // Черно-белый, более    сложный
режим
        // Вначале рисуем    изображение в
буфере цвета        glDrawPixels(iWidth,
iHeight, eFormat, GL_UNSIGNED_BYTE, pImage);

```

```

        // Распределяем память для карты яркости
        pModifiedBytes = (GLbyte *)malloc(iWidth
* iHeight);

        // Масштабируем цвета согласно стандарту
NSTC

        glPixelTransferf(GL_RED_SCALE, 0.3f);
        glPixelTransferf(GL_GREEN_SCALE, 0.59f);
        glPixelTransferf(GL_BLUE_SCALE, 0.11f);

        // Считываем пиксели в буфер (будем при-
менено увеличение)
        glReadPixels(0,0,iWidth, iHeight,
GL_LUMINANCE, GL_UNSIGNED_BYTE, pModifiedBytes);

        // Масштабирование цвета возвращается в
норму

        glPixelTransferf(GL_RED_SCALE, 1.0f);
        glPixelTransferf(GL_GREEN_SCALE, 1.0f);
        glPixelTransferf(GL_BLUE_SCALE, 1.0f);
        break;

    case 8:        // Инверсия цветов
        invertMap[0] = 1.0f;
        for(i = 1; i < 256; i++)
            invertMap[i] = 1.0f - (1.0f / 255.0f
* (GLfloat)i);

        glPixelMapfv(GL_PIXEL_MAP_R_TO_R, 255,
invertMap);
        glPixelMapfv(GL_PIXEL_MAP_G_TO_G, 255,
invertMap);
        glPixelMapfv(GL_PIXEL_MAP_B_TO_B, 255,
invertMap);
        glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
        break;

    case 1:        // Просто копия старого изображе-
ния

    default:

        // Данная строка специально ос-
тавлена пустой

```

```

        break;
    }

    // Рисуются пиксели
    if(pModifiedBytes == NULL)
        glDrawPixels(iWidth, iHeight, eFormat,
GL_UNSIGNED_BYTE, pImage);
    else
    {
        glDrawPixels(iWidth, iHeight, GL_LUMINANCE,
GL_UNSIGNED_BYTE, pModifiedBytes);
        free(pModifiedBytes);
    }

// Обновление всего до настроек по умолчанию
glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
glPixelTransferf(GL_RED_SCALE, 1.0f);
glPixelTransferf(GL_GREEN_SCALE, 1.0f);
glPixelTransferf(GL_BLUE_SCALE, 1.0f);
glPixelZoom(1.0f, 1.0f); //Без
увеличения пикселей

// Переключает буферы
glutSwapBuffers();
}

void ChangeSize(int w, int h)
{
// Предотвращает деление на нуль, когда окно слишком
маленькое
// (нельзя сделать окно нулевой ширины)
    if(h == 0)
        h = 1;

    glViewport(0, 0, w, h);

// Система координат обновляется перед модификацией
glMatrixMode(GL_PROJECTION);

```

```

        glLoadIdentity();

// Устанавливается объем отсечения
        gluOrtho2D(0.0f, (GLfloat) w, 0.0, (GLfloat) h);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

////////////////////////////////////
////////////////////////////////////
// Точка входа основной программы

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GL_DOUBLE);
    glutInitWindowSize(800 ,600);
    glutCreateWindow("OpenGL Image Operations");
    glutReshapeFunc(ChangeSize);
    glutDisplayFunc(RenderScene);

// Создается меню и добавляются опции выбора
    glutCreateMenu(ProcessMenu);
    glutAddMenuEntry("Save Image",0);
    glutAddMenuEntry("Draw Pixels",1);
    glutAddMenuEntry("Flip Pixels",2);
    glutAddMenuEntry("Zoom Pixels",3);
    glutAddMenuEntry("Just Red Channel",4);
    glutAddMenuEntry("Just Green Channel",5);
    glutAddMenuEntry("Just Blue Channel",6);
    glutAddMenuEntry("Black and White", 7);
    glutAddMenuEntry("Invert Colors", 8);
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    SetupRC();           // Настройка

    glutMainLoop();      // Основной программный
        цикл

    ShutdownRC();        // Выключение

```



```

    return 0;
}

```

Основной каркас программы достаточно прост. В отличие от предыдущего примера сейчас изображение загружается и содержится в памяти все время работы программы, поэтому при каждом изменении экрана не требуется повторная загрузка изображения. Как показано ниже, информация об изображении и указатель на требуемые байты хранятся как глобальные переменные модуля.

```

    static GLubyte *plmage = NULL;
    static GLint lWidth, iHeight, iComponents;
    static GLenum eFormat;

```

Таким образом, функция `SetupRC` всего лишь загружает изображение и инициализирует глобальные переменные, содержащие формат, ширину и высоту изображения.

```

    // Загружаем изображение лошади
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    plmage = gltLoadTGA("horse.tga", SiWidth,
SiHeight,
    SiComponents, SeFormat);

```

При завершении программы память, которая ей была выделена с помощью функции

```

    gltLoadTGA в ShutdownRC, освобождается
free(plmage);

```

В главной функции создается меню и добавляются позиции и значения для различных требуемых операций

```

    // Создается меню и добавляются опции выбора
glutCreateMenu(ProcessMenu);
glutAddMenuEntry("Save Image", 0);
glutAddMenuEntry("Draw Pixels", 1);
glutAddMenuEntry("Flip Pixels", 2);
glutAddMenuEntry("Zoom Pixels", 3);
glutAddMenuEntry("Just Red Channel", 4);
glutAddMenuEntry("Just Green Channel", 5);
glutAddMenuEntry("Just Blue Channel", 6);
glutAddMenuEntry("Black and White", 7);
glutAddMenuEntry("Invert Colors", 8);
glutAttachMenu(GLUT_RIGHT_BUTTON);

```

Эти позиции меню устанавливают желаемое значение переменной `iRenderMode` или, если значение равно 0, записывают изображение так, как оно отображено в данный момент

```
void ProcessMenu(int value)
{
    if(value == 0)
        // Записываем изображение
        gltWriteTGA("ScreenShot1.tga");
    else
        // Меняем индекс режима визуализации на индекс,
        // соответствующий позиции меню
        iRenderMode = value;
    // Активируем перерисовывание изображения
    glutPostRedisplay();
}
```

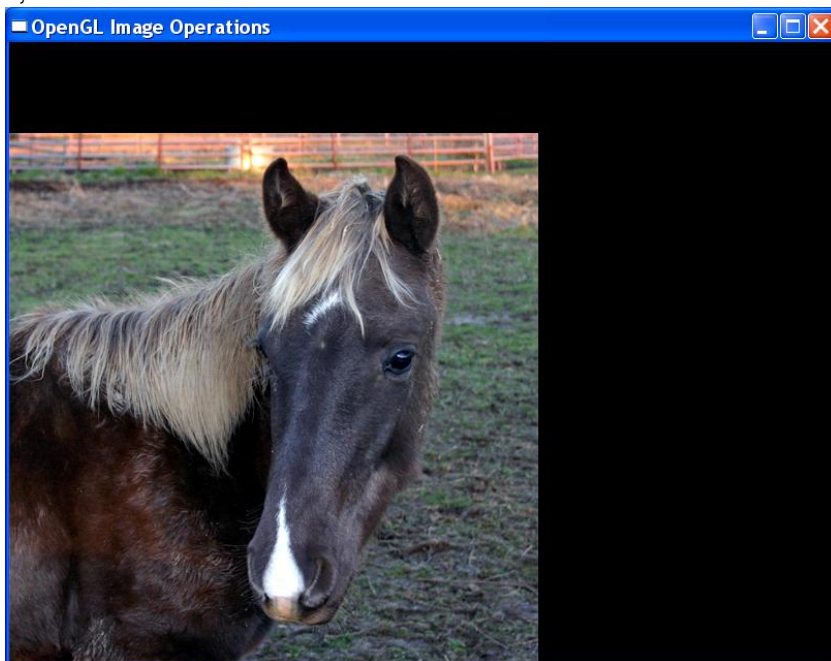


Рис.5. – Результат выполнения по умолчанию листинга 5

Наконец, изображение рисуется в буфере цвета, и за это отвечает функция `RenderScene`. Данная функция содержит оператор выбора, использующий переменную `iRenderMode` для выбора одного из

восьми различных режимов рисования. По умолчанию просто вызывается неизменная функция `glDrawPixels`, помещающая изображение в левом нижнем углу окна, как показано на рис. 5. Другие случаи рассмотрены ниже.

## Изменение масштаба пикселей

Другой простой, но достаточно распространенной операцией, которую часто требуется производить с данными пикселей, является сжатие или растягивание изображения. В контексте OpenGL это называется *масштабированием пикселей* (pixel zoom); выполняется оно с помощью следующей функции:

```
void glPixelZoom(GLfloat xfactor, GLfloat yfactor);
```

Аргументы `xfactor` и `yfactor` задают коэффициенты масштабирования по осям *x* и *y*). С помощью указанной операции можно сжимать, растягивать и даже переворачивать изображение. Например, при коэффициенте 2 изображение будет записано с двукратным размером вдоль заданной оси, а при коэффициенте 0.5 оно будет сжато вдвое.



Рис.6. – Масштабирование пикселей для растягивания изображения на все окно

Для примера выбор в программе OPERATIONS позиции Zoom Pixels из меню соответствует третьему режиму визуализации. В этом случае перед вызовом функции `glDrawPixels` выполняются приведенные ниже строки кода, задающие коэффициенты масштабирования в направлениях *x* и *y* такими, чтобы растянутое изображение заняло все окно.

```
case 3:  
    // Увеличиваем пиксели для заполнения окна
```

```

glGetIntegerv(GL_VIEWPORT, iViewport);
glPixelZoom((GLfloat) iViewport[2] / (GLfloat)iwidth,
(GLfloat) iViewport[3] / (GLfloat)iHeight);
break;

```

Результат выполнения программы для этого случая показан на рис.

6.

С другой стороны, отрицательный коэффициент масштабирования переворачивает изображение вдоль выбранной оси. Использование такого коэффициента не только обращает порядок пикселей на изображении, но и меняет на обратное направление рисования пикселей относительно растрового положения. Например, при обычном рисовании изображения левый нижний угол помещен в текущее растровое положение. Если оба коэффициента масштабирования отрицательные, растровое положение становится правым верхним углом полученного изображения.

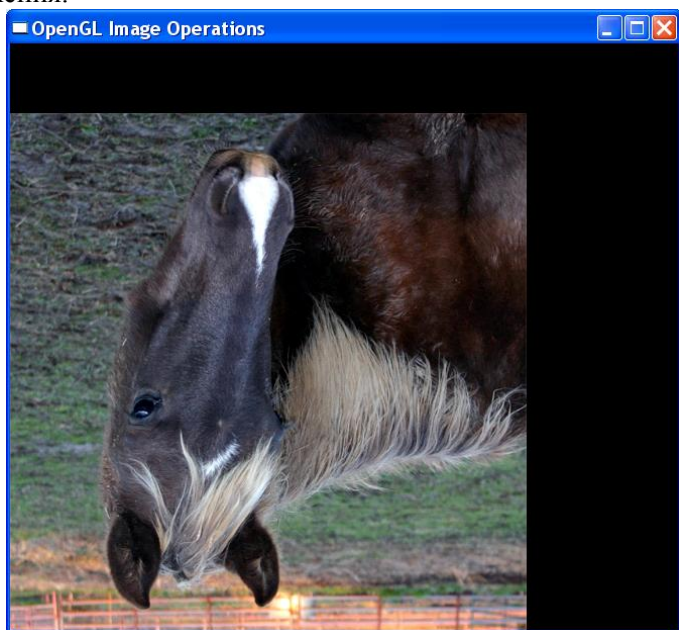


Рис.7. – Изображение, полученное с помощью инверсии по осям  $x$  и  $y$

В программе выбор из меню позиции Flip Pixels переворачивает изображение горизонтально и вертикально. Как видно из приведенно-

го фрагмента кода, оба масштабных коэффициента пикселей устанавливаются равными -1.0, и растровое положение меняется с левого нижнего угла окна на правый верхний угол (задается шириной и высотой изображения).

```
case 2:
```

```
// Переворачиваем пиксели
```

```
glPixelZoom(-1.Of, -1.Of);
```

```
glRasterPos2i(iWidth, iHeight);
```

```
break;
```

Инвертированное изображение, получающееся при выборе этой опции из меню, показано на рис. 7.

## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Воспроизвести результаты, представленные в теоретическом обзоре, научиться работать с битовыми масками и пиксельными изображениями, согласно варианту, выполнить преобразование цветового пространства и расположения изображения.

### ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту. По завершении готовится отчет.

- 1) Для [Листинга 1](#) создать битовые образы согласно варианту
- 2) Для [Листингов 2, 3, 4](#) осуществить вывод изображения в формате tga согласно варианту
- 3) Для [Листинга 5](#) реализовать преобразование изображения согласно варианту.

### ВАРИАНТЫ ЗАДАНИЙ

Задание 1

- 1) Создать свой битовый образ (не из примера) размером 34x34. Отобразить на окне 20 битовых образов с произвольной координатой.
- 2) Создать два разных битовых образа (не из примера) размерами 20x20 и 16x16. Отобразить построчно в 8 строк с чередованием. Образов в строке должно быть 8. Цвета каждого образа должны быть разными.
- 3) Создать свой битовый образ (не из примера) размером 24x24. Отобразить битовые образы в центре окна в виде квадрата размером 6x6. Цвет каждого изображения должен быть задан случайно.
- 4) Создать свой битовый образ (не из примера) размером 16x36. Отобразить 10 строк образов со случайным расстоянием между строками. Цвет каждой строки образов должен быть разным.
- 5) Создать свой битовый образ (не из примера) размером 24x24. Отобразить 10 строк образов с уменьшением количества образов на 1 в каждой строке. Первоначально в первой строке 10 образов.

Выравнивание образов в строке должно быть по центру окна. В итоге должен получиться треугольник.

- 6) Создать свой битовый образ (не из примера) размером 20x20. Отобразить 10 строк образов с уменьшением количества образов на 1 в каждой строке. Первоначально в первой строке 12 образов. Выравнивание образов в строке должно быть по ширине окна.
- 7) Создать два разных битовых образа (не из примера) размерами 16x20 и 10x10. Отобразить два образа в виде песочных часов. В верхней части окна до центра окна один образ с уменьшением количества элементов в строке, в нижней части – другой. Минимальное количество образов максимальной строке 10. Цвета каждого образа должны быть разными.
- 8) Создать два разных битовых образа (не из примера) размерами 10x10 и 15x15. Отобразить ромб, в котором границей является образ №1, а заполнение происходит при помощи образа №2. Граница должна быть одного цвета, заполненные внутренние образы случайного цвета.
- 9) Создать два разных битовых образа (не из примера) размерами 20x20 и 12x12. В координатных четвертях в виде квадрата 6x6 (отобразить соответствующие оси) вывести образ №1 одного цвета, образ №2 другого цвета, образы №1 и 2 одного цвета в случайной последовательности и образы №1 и 2 случайного цвета в случайной последовательности.
- 10) Создать свой битовый образ (не из примера) размером 40x40. Вывести его в виде квадрата размером 6x6 по центру окна. Каждый образ должен иметь случайный цвет и должен быть повернут на произвольный угол кратный 90 градусам, а также зеркально отражен.

## Задание 2

- 1) Для собственного изображения формата tga, осуществить его вывод координатные четверти.
- 2) Для собственного изображения формата tga, осуществить его вывод (4 раза) от верхнего левого до правого нижнего угла.
- 3) Для собственного изображения формата tga, осуществить его вывод (4 раза). Исходное изображение и по компонентам RGB. Вывод начинается от правого верхнего до левого нижнего угла.

- 4) Создать два собственных изображения формата tga. Осуществить их вывод (6 изображений по 3) с разными компонентами RGB.
- 5) Создать два собственных изображения формата tga. Заполнить ими окно по координатным четвертям.
- 6) Для собственного изображения формата tga, осуществить 5 его выводов каждый раз обрезая высоту на 20%.
- 7) Для собственного изображения формата tga, осуществить 4 его вывода каждый раз увеличивая высоту на 20%. Изначально отображается только 20% рисунка.
- 8) Для собственного изображения формата tga, отобразить 10 его копий со случайными координатами.
- 9) Создать два собственных изображения формата tga. Одно изображение вывести от левого верхнего к правому нижнему углу, другое от правого верхнего к левому нижнему. В центре окна отобразить небольшой синий квадрат.
- 10) Создать три собственных изображения формата tga. Вывести их в три столбца по 4 изображения в каждом.

### Задание 3

- 1) Для собственного изображения формата tga выполнить поворот на 90 градусов по часовой стрелке и отобразить только R и G компоненты.
- 2) Для собственного изображения формата tga выполнить зеркальное отражение относительно горизонтали и перевести в черно-белый формат.
- 3) Для собственного изображения формата tga выполнить увеличение по оси x в два раза, по оси y в 0.7 раза. Отобразить инвертированное изображение.
- 4) Для собственного изображения формата tga выполнить инвертирование цветов только красной компоненты.
- 5) Для собственного изображения формата tga выполнить зеркальное отражение относительно вертикали и отобразить красную и зеленую компоненты.
- 6) Для собственного изображения формата tga выполнить каждый раз при запуске произвольное масштабирование в диапазоне от 0.5 до 3.0 по каждой из осей. Каждый раз отображать только одну из компонент RGB.



- 7) Для собственного изображения формата tga реализовать не менее трех разных переводов в градации серого (использовать меню).
- 8) Для собственного изображения формата tga выполнить случайное изменение компонент RGB при каждом запуске.
- 9) Для собственного изображения формата tga выполнить поворот на 90 градусов против часовой стрелки и увеличить масштаб по осям на 1.2.
- 10) Для собственного изображения формата tga выполнить инвертирование изображения в градациях серого. Первоначально изображение по компонентам RGB имеет следующие составляющие 0.6, 0.9, 0.78.

## КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Дайте определение битового и растрового образов.
2. Сформулируйте требования предъявляемые к битовым и пиксельным образам.
3. Классифицируйте пиксельные форматы поддерживаемые OpenGL.
4. Обоснуйте необходимость и опишите процесс упаковки описания цвета пикселя.
5. Охарактеризуйте процесс передачи пикселей.
6. Перечислите и раскройте основные цели параметров передачи пикселей.
7. Раскройте значение термина перемещение пикселей.
8. Опишите роль механизма отображения цвета.
9. Сформулируйте алгоритм действий для сохранения размеров передаваемого изображения.
10. Сформулируйте роль функции `glBitmap` и её параметров.
11. Выделите ключевые особенности формата `targa`, используемого для хранения изображений.
12. Предложите варианты масштабирования пикселей.
13. Охарактеризуйте механизм отражения пиксельного изображения.
14. Перечислите параметры, ожидаемые функцией `glPixelStorei`.
15. Классифицируйте типы данных, используемые в OpenGL для описания цвета. и их соответствия в C-подобных языках.

## ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 2 занятия (4 академических часа: 3 часа на выполнение и сдачу практического задания и 1 час на подготовку отчета).

Номер варианта студента назначается индивидуально преподавателем.

В отчете должны быть представлены:

- 1) Текст задания для лабораторной работы и номер варианта.
- 2) Листинги программ для заданий 1 – 3. В отчете должны быть представлены все полные листинги программ. При необходимости листинги программ можно сокращать, если повторные части кода присутствуют в ранее указанных листингах. Во всех листингах программ должны быть подробные комментарии к основным функциям приложения. Выполнение задания должно сопровождаться снимками экрана.

Отчет по каждому новому заданию начинать с новой страницы. В выводах отразить затруднения при ее выполнении и достигнутые результаты.

Отчет на защиту предоставляется в печатном виде.

Отчет содержит: Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы, результаты выполнения работы, выводы.

## ОСНОВНАЯ ЛИТЕРАТУРА

1. Боресков А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов - Издательство "ДМК Пресс", 2010. - 232 с. - ISBN 978-5-94074-578-5; ЭБС «Лань». - URL: [https://e.lanbook.com/book/1260#book\\_name](https://e.lanbook.com/book/1260#book_name) (23.12.2017).
2. Васильев С.А. OpenGL. Компьютерная графика : учебное пособие / С.А. Васильев. — Электрон. текстовые данные. — Тамбов: Тамбовский государственный технический университет, ЭБС АСВ, 2012. — 81 с. — 2227-8397. — Режим доступа: <http://www.iprbookshop.ru/63931.html> — ЭБС «IPRbooks», по паролю
3. Вольф Д. OpenGL 4. Язык шейдеров. Книга рецептов/ Вольф Д. - Издательство "ДМК Пресс", 2015. - 368 с. - 978-5-97060-255-3; ЭБС «Лань». - URL: [https://e.lanbook.com/book/73071#book\\_name](https://e.lanbook.com/book/73071#book_name) (23.12.2017).
4. Гинсбург Д. OpenGL ES 3.0. Руководство разработчика/Д. Гинсбург, Б. Пурномо. - Издательство "ДМК Пресс", 2015. - 448 с. - ISBN 978-5-97060-256-0; ЭБС «Лань». - URL: [https://e.lanbook.com/book/82816#book\\_name](https://e.lanbook.com/book/82816#book_name) (29.12.2017).
5. Лихачев В.Н. Создание графических моделей с помощью Open Graphics Library / В.Н. Лихачев. — Электрон. текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 201 с. — 2227-8397. — Режим доступа: <http://www.iprbookshop.ru/39567.html>
6. Забелин Л.Ю. Основы компьютерной графики и технологии трехмерного моделирования : учебное пособие/ Забелин Л.Ю., Конюкова О.Л., Диль О.В.— Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2015.— 259 с.— Режим доступа: <http://www.iprbookshop.ru/54792>.— ЭБС «IPRbooks», по паролю
7. Папуловская Н.В. Математические основы программирования трехмерной графики : учебно-методическое пособие / Н.В. Папуловская. — Электрон. текстовые данные. — Екатеринбург: Уральский федеральный университет, 2016. — 112 с. — 978-5-7996-1942-8. — Режим доступа: <http://www.iprbookshop.ru/68345.html>
8. Перемитина, Т.О. Компьютерная графика : учебное пособие / Т.О. Перемитина ; Министерство образования и науки Российской Федера-

ции, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). - Томск : Эль Контент, 2012. - 144 с. : ил.,табл., схем. - ISBN 978-5-4332-0077-7 ; - URL: <http://biblioclub.ru/index.php?page=book&id=208688> (30.11.2017).

## ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

### Электронные ресурсы:

1. [http://opengl-master.ru/view\\_post.php?id=69](http://opengl-master.ru/view_post.php?id=69) - Растровые изображения в OpenGL
2. [http://opengl-master.ru/view\\_post.php?id=64](http://opengl-master.ru/view_post.php?id=64) - Пиксельные образы в OpenGL
3. [http://aco.ifmo.ru/el\\_books/computer\\_visualization/lectures/7.html](http://aco.ifmo.ru/el_books/computer_visualization/lectures/7.html) - Визуализация растровых примитивов
4. [http://www.codenet.ru/progr/opengl/opengl\\_05.php](http://www.codenet.ru/progr/opengl/opengl_05.php) - OpenGL - Работа с картинками