

ЯРОШЕВИЧ В.А.

3D МОДЕЛИРОВАНИЕ

Лекции, практические занятия, лабораторный практикум

Версия 1.15 (09.11.2018)

5-ый курс

МИЭТ — 2018

Оглавление

1 Введение в шейдеры	4
1.0.1 OpenGL Shading Language	4
1.0.2 Доступ к последним возможностям OpenGL посредством GLEW	5
1.0.3 Математическая библиотека GLM	6
1.0.4 Компиляция шейдера	6
1.0.5 Сборка программы шейдера	8
1.0.6 Передача данных в шейдер с помощью вершинных атрибутов и буферов	11
1.0.7 Передача данных в шейдер с помощью переменных uniform	12
1.0.8 Получение списка активных переменных uniform	14
1.0.9 Шейдеры (Shaders)	14
2 Текстуры в шейдерах	16
2.1 Двумерные текстуры	16
2.2 Несколько текстур	19
2.3 α -карты для создания дырок и отверстий	21
2.4 Карты нормалей	22
2.5 Имитация зеркального отражения с помощью кубических карт	25
3 Мозаичные шейдеры	27
3.1 Геометрические и мозаичные шейдеры	27
3.2 Геометрический шейдер	27
3.3 Разбиение кривой	28
3.4 Тесселяция (замощение) плоского четырёхугольника	32
4 Эффекты с помощью шейдеров	40
4.0.1 Эффект ряби (Ripple Effect)	40
4.0.2 Эффект волны (Wave Effect)	41
4.0.3 Эффект тиснения (Emboss Effect using sprite and alpha)	41
4.0.4 Эффект ударной волны (Shockwave Effect)	42
4.0.5 Эффект телевизора (TV Effect)	42
4.0.6 Эффект яркости/выцветания (Brightness/Bloom Effect)	44
4.0.7 Рельефное текстурирование «Normal Mapping»	45
4.0.8 Эффект анимации травы (Grass Animation Effect)	46

Оглавление	3
4.0.9 Эффект голограммы (Hologram Effect)	47
4.0.10 Эффект воды для TDS.	47
4.0.11 Эффект Гауссова размытия (Gaussian Blur Effect)	48
4.0.12 Эффект старого фильма / Сепии (Old Film Effect)	49
4.0.13 Радиальное размытие (Radial Blur)	49
4.0.14 Эффект воды для платформера	52
4.0.15 Эффект лучей проходящих сквозь препятствия	54
4.0.16 Эффект стекла / витража (Stained Glass Shader)	55
4.0.17 Эффект светлячков/блёстков/сияний (Happy Fireflies)	56
4.0.18 Эффект огненного вакуума/фаерболла (Fire Vacuum)	57
4.0.19 Эффект подводного освещения	58
4.0.20 Мультитекстурирование (Multi Texturing)	59
4.0.21 Эффект черно-белого изображения (Grey Scale Effect)	59
4.0.22 Эффект мозаики / пикселизации (Mosaic Effect)	60
4.0.23 Эффект линзы / «рыбий глаз» (Magnify Effect).	61
5 Рельефное текстурирование	63
5.0.1 Рельефное текстурирование	63
5.0.2 Построение рельефного текстурирования	63
5.0.3 Касательное пространство	64
5.0.4 Вычисление касательных векторов	65
5.0.5 Реализация шейдера	68
6 Моделирование жидкости и ткани	70
6.1 Имитация жидкости	70
6.1.1 Имитация поверхности жидкости	70
6.1.2 Уравнение волны	70
6.1.3 Расчёт смещения поверхности	73
6.1.4 Реализация	74
6.2 Имитация ткани	77
6.2.1 Система пружин	77
6.2.2 Внешние силы	79
6.2.3 Реализация	79
6.3 Упражнения	80
7 Лабораторный практикум	81
7.1 Лабораторная работа 1. Знакомство с шейдерами.	81
7.2 Лабораторная работа 2. Рельефное текстурирование.	81
7.3 Лабораторная работа 3. Эффект тени.	81
7.4 Лабораторная работа 4. Математические модели объектов.	82
7.5 Лабораторная работа 5. Работа с физическим процессором.	82
Литература	83

Глава 1

Введение в шейдеры

OpenGL Shading Language (GLSL) версии 4.0 даёт разработчикам небывалую мощь и гибкость при создании современных интерактивных программ. Появляется возможность напрямую и в полной мере использовать силу современных *графических процессоров* (*Graphics Processing Units* или *GPUs*) благодаря мощному языку и API (англ. *Application Programming Interface* или *интерфейс программирования приложений*, *интерфейс прикладного программирования*). Программу на GLSL не существуют сами по себе, а входят как составная часть в большую программу для OpenGL.

1.0.1 OpenGL Shading Language

На сегодняшний день OpenGL Shading Language (GLSL) является основной составной частью OpenGL API. Забегая вперёд скажем, что всякая программа для OpenGL внутри себя использует одну или несколько GLSL-программ. Эти минипрограммы на GLSL называются *шейдерами*. Программа-шейдер исполняется в графическом процессоре и реализует алгоритмы эффектов освещенности и закраски для 3-мерных изображений. При этом программа-шейдер способна на большее, чем реализация только алгоритма закраски поверхностей. Они также могут создавать анимацию, мозаику и даже общие математические вычисления.

Программа-шейдер предназначена к исполнению прямо в графическом процессоре и часто с применением параллельных вычислений. Например фрагментный шейдер исполняется один раз для каждого пикселя, а вычисления запускаются одновременно в разных потоках графического процессора. Число процессоров в графической подсистеме определяет число одновременных потоков. Всё это делает программы-шейдеры необычайно эффективными и позволяет программисту очень легко задействовать параллельные вычисления.

Вычислительная мощность современных графических подсистем впечатляет. В следующей таблице приведены количества шейдер-процессоров в некоторых моделях плат NVIDIA GeForce 900 (исотчик: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units).

Модель	Шейдерные процессоры
GeForce GTX 950	768
GeForce GTX 960	1024
GeForce GTX 970	1664
GeForce GTX 980	2048
GeForce GTX 980 Ti	2816
GeForce GTX Titan X	3072

1.0.2 Доступ к последним возможностям OpenGL посредством GLEW

Файлы `gl.h` и `opengl32.lib`, поставляемые вместе с компиляторами от Microsoft соответствуют версии OpenGL 1.1. Для доступа к современным возможностям OpenGL в Windows приходится подключать дополнительную библиотеку GLEW (OpenGL Extension Wrangler, <http://glew.sourceforge.net>).

В проект нужно добавить файлы `glew.h` и `wglew.h`, а также подменить файл `glew32.lib`. Пример использования:

```
#include <GL/glew.h> // этот файл должен идти первым в списке!
#include <GL/gl.h>
#include <GL/glu.h>

// перед вызовом любой функции OpenGL обязательно вызвать:
GLenum err = glewInit();
if( GLEW_OK != err )
{
    fprintf(stderr, "Error initializing GLEW: %s\n",
    glewGetErrorString(err) );
}
```

Утилиты библиотеки GLEW

В библиотеке GLEW есть очень полезные утилиты.

GLEW visualinfo После запуска утилиты `visualinfo` появляется файл `visualinfo.txt` со списком всех доступных расширений OpenGL, WGL (OpenGL Extension for Microsoft Windows) и GLU (OpenGL Utility Library), а также графических параметров (формат пикселей, доступность буфера пикселей и т.д.).

GLEW glewinfo После запуска утилиты `glewinfo` появляется файл `glewinfo.txt` со списком всех функций, поддерживаемых драйвером.

Проверка доступности расширения

Доступность расширений проверяется через глобальные переменные библиотеки GLEW. Например, чтобы проверить доступность `ARB_vertex_program`, нужно написать следующее:

```
if ( ! GLEW_ARB_vertex_program ) {
    fprintf(stderr, "Расширение ARB_vertex_program отсутствует!\n");
    ...
```

}

1.0.3 Математическая библиотека GLM

В основе компьютерной графики лежат математические расчёты. В ранних версиях OpenGL поддерживала матричный стек и операции с матрицами (`GL_MODELVIEW` и `GL_PROJECTION`). Начиная с версии OpenGL 4.0 этой возможности больше нет. Поэтому задача подготовки матриц преобразований и проекций решается на усмотрение программиста. Возникает потребность в библиотеке, которая умеет работать с матрицами и векторами.

Одной из таких библиотек является GLM (OpenGL Mathematics). Синтаксис библиотеки приближен к языку GLSL. Кроме того в библиотеке есть функции, которых теперь так не достаёт в OpenGL, например: `glOrtho`, `glRotate` или `gluLookAt`.

Библиотека доступна на странице: <http://glm.g-truc.net>. Далее приведём небольшой пример пользования библиотекой:

```
#include <glm/glm.hpp> // подключение основных функций GLM
#include <glm/gtc/matrix_transform.hpp> // матричные преобразования
#include <glm/gtx/transform2.hpp> // дополнительное расширение
// Классы GLM находятся в пространстве имён glm
// Пример использования:
glm::vec4 position = glm::vec4( 1.0f, 0.0f, 0.0f, 1.0f );
glm::mat4 view = glm::lookAt( glm::vec3(0.0, 0.0, 5.0),
                             glm::vec3(0.0, 0.0, 0.0),
                             glm::vec3(0.0, 1.0, 0.0) );
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate( model, 90.0f, glm::vec3(0.0f, 1.0f, 0.0) );
glm::mat4 mv = view * model;
glm::vec4 transformed = mv * position;
```

1.0.4 Компиляция шейдера

Компилятор GLSL встроен в библиотеку OpenGL. Программы-шейдеры могут компилироваться только внутри работающей OpenGL-программы. Начиная с версии OpenGL 4.1 появилась возможность сохранять скомпилированные программы-шейдеры в файл для ускорения работы основной OpenGL-программы.

Компиляция состоит в создании шейдер-объекта, передачи исходного кода (в виде строки) в шейдер-объект и самой компиляции кода. Процесс изображен на рис. 0.1.

Для демонстрации создадим простейший шейдер в файле `basic.vert`:

```
#version 400
in vec3 VertexPosition;
in vec3 VertexColor;
out vec3 Color;
void main() {
    Color = VertexColor;
    gl_Position = vec4( VertexPosition, 1.0 );
}
```

Шейдер не несёт никакой полезной нагрузки. На вход шейдера поступают атрибуты `VertexPosition` и `VertexColor` и без изменения передаются в выходные переменные `gl_Position` и `Color`.

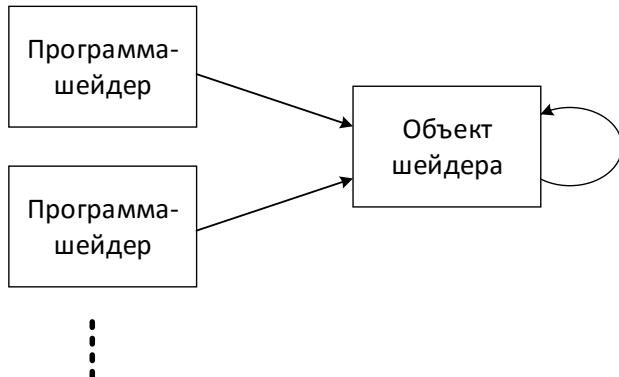


Рис. 0.1: Компиляция шейдера.

Далее нам нужно написать базовую программу для OpenGL, например, на основе функций из GLUT. Потом загружаем исходный код шейдера в массив символов `shaderCode`. Очень важно добавить нулевой символ в конце. В примере ниже мы предполагаем, что переменная `shaderCode` указывает на массив `GLchar`, который корректно заканчивается нулевым символом.

Для компиляции шейдера необходимы следующие шаги:

```

// 1. Создаём объект шейдера.
GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );
if( 0 == vertShader ) {
    fprintf(stderr, "Error creating vertex shader.\n");
    exit(1);
}

// 2. Копируем исходный код (возможно, из разных файлов) в один объект шейдера.
const GLchar* shaderCode = loadShaderAsString("basic.vert");
const GLchar* codeArray[] = { shaderCode };
glShaderSource( vertShader, 1, codeArray, NULL );

// 3. Компилируем шейдер.
glCompileShader( vertShader );

// 4. Проверяем успешность компиляции
GLint result;
glGetShaderiv( vertShader, GL_COMPILE_STATUS, &result );
if( GL_FALSE == result ) {
    fprintf( stderr, "Не удалось скомпилировать вершинный шейдер!\n" );
    GLint logLen;
    glGetShaderiv( vertShader, GL_INFO_LOG_LENGTH, &logLen );
    if( logLen > 0 ) {
        char * log = (char *)malloc(logLen);
        GLsizei written;
        glGetShaderInfoLog(vertShader, logLen, &written, log);
        fprintf(stderr, "Лог шейдера:\n%s", log);
        free(log);
    }
}
  
```

Первый делом мы создали объект для шейдера с помощью функции `glCreateShader`. Аргументом функции является тип шейдера, возможные значения: `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, `GL_TESS_EVALUATION_SHADER` или `GL_TESS_CONTROL_SHADER`. В нашем примере используется вершинный шейдер, поэтому выбираем `GL_VERTEX_SHADER`. Функция возвращает значение («указатель»),

необходимое для обращения к объекту шейдера. Значение сохраняется в переменной `vertShader`. Если при компиляции происходит ошибка, то функция возвращает 0. Мы проверяем этот случай и выводим на экран необходимое сообщение.

После создания объекта шейдера мы загружаем исходный код в объект шейдера с помощью функции `glShaderSource`. Последняя функция принимает на вход массив строк, чтобы можно было компилировать одновременно несколько файлов. Поэтому перед вызовом `glShaderSource` мы помещаем указатель на наш исходный код в массив `sourceArray`. Первый аргумент `glShaderSource` — это «указатель» на объект шейдера. Во втором аргументе число строк с исходными кодами шейдеров, которые содержатся в массиве. Третий аргумент указывает на массив строк исходного кода. Последний аргумент — это массив значений `GLint`, содержащий длины строк в массиве. В нашем примере передаётся значение `NULL`. Это означает, что строки с исходным кодом заканчиваются нулевым символом. Когда функция возвращает управление в основную программу, исходный код шейдеров уже скопирован во внутреннюю память OpenGL. Поэтому память, использованную для исходного кода, можно очистить.

Следующий шаг — это компиляция шейдера. Мы делаем это, просто вызвав `glCompileShader` с «указателем» на объект шейдера. Ясно, что компиляция зависит от наличия ошибок в исходном коде шейдера. Поэтому следующим шагом мы проверяем успешность компиляции.

Результат компиляции можно узнать, вызвав `glGetShaderiv`. Эта функция запрашивает атрибуты объекта шейдера. В нашем случае результату компиляции соответствует константа `GL_COMPILE_STATUS`, которую передаем во втором аргументе. Первый аргумент — «указатель» на объект шейдера. Третий аргумент — указатель на целое число, куда будет записан результат: либо `GL_TRUE`, либо `GL_FALSE`.

1.0.5 Сборка программы шейдера

Если наши шейдеры уже скомпилированы, то перед встраиванием их в конвейер OpenGL нам нужно собрать (*link*) из программу шейдера. Среди прочего во время сборки устанавливаются связи между входными одного шейдера и выходными переменными другого, а также связи между входными и выходными переменными шейдера и программой OpenGL (рис.,0.3).

Сборка состоит из шагов, похожих на этапы его компиляции. Мы прикрепляем все объекты шейдеров к новому объекту программы шейдера. Потом говорим объекту программы шейдера произвести сборку.

Предположим, что у нас уже есть два скомпилированных объекта шейдеров, чьи указатели содержатся в переменных `vertShader` и `fragShader`.

Для фрагментного шейдера будем использовать следующий код:

```
#version 400
in vec3 Color;
out vec4 FragColor;
void main() {
    FragColor = vec4(Color, 1.0);
}
```

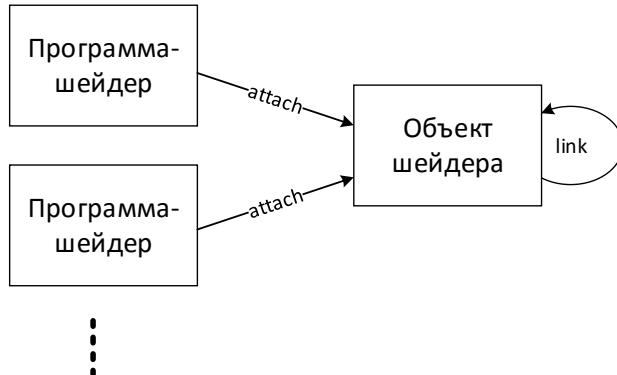


Рис. 0.2: Сборка шейдеров.

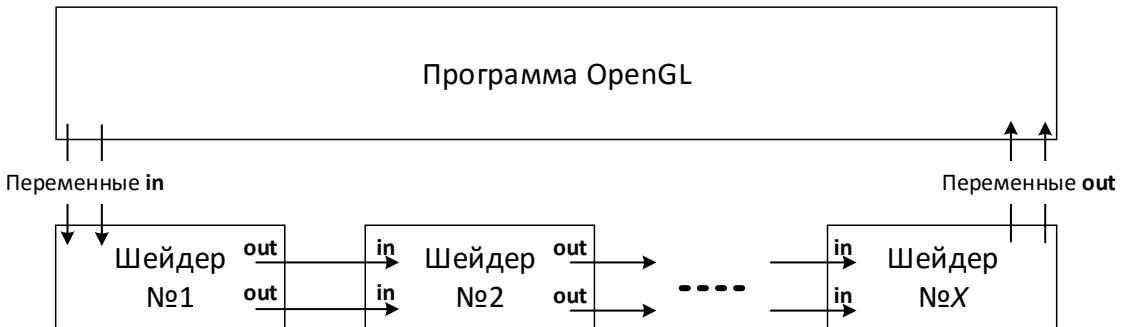


Рис. 0.3: Взаимодействие входных и выходных переменных шейдеров после сборки.

В инициализирующей функции OpenGL после шагов компиляции шейдеров `vertShader` и `fragShader` выполним следующее:

```

// 1. Создаём объект программы
GLuint programHandle = glCreateProgram();
if( 0 == programHandle ) {
    fprintf(stderr, "Ошибка при создании объекта программы.\n");
    exit(1);
}

// 2. Прикрепляем шейдеры к объекту программы.
glAttachShader( programHandle, vertShader );
glAttachShader( programHandle, fragShader );

// 3. Собираем программу.
glLinkProgram( programHandle );

// 4. Проверяем успешность сборки.
GLint status;
glGetProgramiv( programHandle, GL_LINK_STATUS, &status );
if( GL_FALSE == status ) {
    fprintf( stderr, "Не удалось собрать программу шейдера!\n" );
    GLint logLen;
    glGetProgramiv( programHandle, GL_INFO_LOG_LENGTH, &logLen );
    if( logLen > 0 ) {
        char * log = (char *)malloc(logLen);
        GLsizei written;
        glGetProgramInfoLog( programHandle, logLen, &written, log );
        fprintf(stderr, "Лог программы: \n%s", log );
        free(log);
    }
}

// 5. Если сборка прошла успешно, встраиваем программу в конвейер OpenGL

```

```
else {
    glUseProgram( programHandle );
}
```

Мы начинаем с вызова `glCreateProgram` для создания пустого объекта программы. В результате мы получим указатель на объект программы `programHandle`. Если произошла ошибка, функция вернёт 0. Мы следим за появлением ошибки и выдаём в этом случае сообщение об ошибке.

Далее мы прикрепляем каждый шейдер к программе с помощью `glAttachShader`. Первый аргумент — это указатель на программу, второй — указатель на прикрепляемый шейдер.

Потом мы собираем программу функцией `glLinkProgram`, куда передаётся указатель на программу. Как в случае компиляции мы проверяем успешность сборки в специальном запросе посредством `glGetProgramiv`. Вторым аргументом теперь будет `GL_LINK_STATUS`. Вывод лога неуспешной сборки происходит аналогично случаю компиляции.

В конце всего после успешной сборки мы встраиваем программу в конвейер OpenGL, вызвав `glUseProgram` и передав указатель на программу.

В примере задаётся треугольник, вершины которого окрашены в красный, зелёный и синий цвета. Внутри треугольника цвета смешиваются за счёт интерполяции (рис. 0.4).

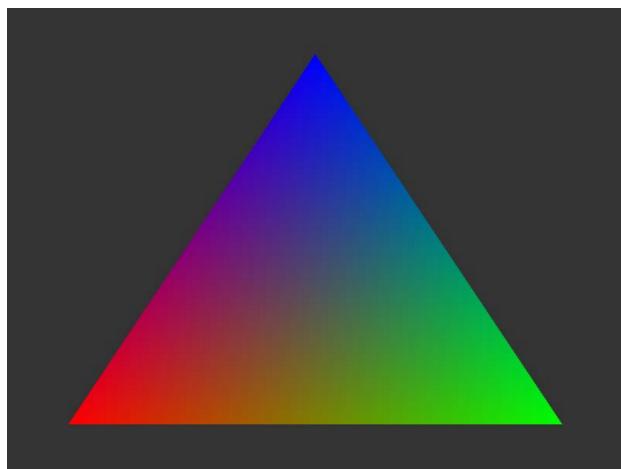


Рис. 0.4: Результат работы программы из примера.

В одной программе OpenGL можно компилировать и собирать много шейдеров. Их можно включать в конвейер OpenGL и исключать из него вызывая `glUseProgram` для выбора желаемой программы шейдеров.

Удалить программу шейдера

Когда программа больше не нужна, её можно удалить из памяти OpenGL, вызвав `glDeleteProgram` с указателем на программу. Это делает указатель нерабочим и освобождает память, использовавшуюся для программы. Отметим, что программа момен-

тально не уничтожается, а помечается для будущего удаления, когда она не будет больше использоваться.

Удаление программы отсоединяет шейдеры, которые были прикреплены к программе, но не уничтожает эти шейдеры, если они не были помечены ранее к удалению функцией `glDeleteShader`.

1.0.6 Передача данных в шейдер с помощью вершинных атрибутов и буферов

Вершинный шейдер вызывается для каждой вершины. Его задача — обработать данные, связанные с вершиной, и передать их (и возможно другую информацию) на следующий этап конвейера. Шейдер будет обрабатывать данные, которые мы должны передать для каждой вершины из программы OpenGL. Среди прочего это прежде всего координаты вершины, нормаль и координаты текстуры. В ранних версиях OpenGL (< 3.0)

```
#version 400
in vec3 VertexPosition;
in vec3 VertexColor;
out vec3 Color;
void main() {
    Color = VertexColor;
    gl_Position = vec4(VertexPosition, 1.0);
}
```

```
#version 400
in vec3 Color;
out vec4 FragColor;
void main() {
    FragColor = vec4(Color, 1.0);
}
```

```
// 1. Перед сборкой программы шейдера зададим соответствие атрибутов вершин и
// выходных переменных шейдера с помощью glBindAttribLocation

// Привязем индекс 0 к входной переменной шейдера "VertexPosition"
glBindAttribLocation(programHandle, 0, "VertexPosition");
// Привязем индекс 1 к входной переменной шейдера "VertexColor"
glBindAttribLocation(programHandle, 1, "VertexColor");

// 2. Создаём глобальную переменную для хранения указателя на объект массива вершин:
GLuint vaoHandle;

// 3. Внутри функции инициализации создаём и заполняем объекты вершинных буферов
// для каждого атрибута.
float positionData[] = {
    -0.8f, -0.8f, 0.0f,
    0.8f, -0.8f, 0.0f,
    0.0f, 0.8f, 0.0f };
float colorData[] = {
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f };

// Создаём объекты буфера
GLuint vboHandles[2];
 glGenBuffers(2, vboHandles);
 GLuint positionBufferHandle = vboHandles[0];
 GLuint colorBufferHandle = vboHandles[1];
 // Заполняем буфер положений вершин
```

```

glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), positionData,
GL_STATIC_DRAW);
// Заполняем буфер цветов
glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), colorData,
GL_STATIC_DRAW);

// 4. Создаём объект массива вершин и привязываемся к нему. В объекте хранится
// соответствие между буферами и входными атрибутами

// Создаём и настраиваем объект массива вершин
 glGenVertexArrays(1, &vaoHandle);
 glBindVertexArray(vaoHandle);

// Включаем массив атрибутов вершин
 glEnableVertexAttribArray(0); // Положение вершины
 glEnableVertexAttribArray(1); // Цвет вершины
// Ставим в соответствие индексу 0 буфер положений вершин
 glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
(GLubyte *)NULL );
// Ставим в соответствие индексу 1 буфер цветов
 glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
(GLubyte *)NULL );

// 5. В функции отрисовки сцены привязываемся к объекту массива вершин и
// вызываем glDrawArrays для инициализации отрисовки
 glBindVertexArray(vaoHandle);
 glDrawArrays(GL_TRIANGLES, 0, 3 );

```

Ключевое слово layout

Мы можем избежать вызова `glBindAttribLocation` внутри OpenGL, если будем использовать ключевое слово `layout` внутри шейдера. Например, мы могли бы избавиться от двух вызовов `glBindAttribLocation`, заменив объявление входных переменных в вершинном шейдере на

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;
```

Это говорит сборщику (linker), что `VertexPosition` должен соответствовать порождающему индексу атрибута 0, а `VertexColor` — индексу 1.

Мы можем использовать ключевое слово `layout`, чтобы задать номер цвета для выходных переменных нашего фрагментного шейдера

```
layout (location = 0) out vec4 FragColor;
```

Это подскажет сборщику связать выходную переменную `FragColor` с цветом 0 без необходимости вызывать `glBindFragDataLocation` внутри программы OpenGL.

1.0.7 Передача данных в шейдер с помощью переменных uniform

Один способ передачи данных в шейдеры — это атрибуты вершины, второй способ — это переменные `uniform`. Такие переменные удобны для передачи данных, общих для многих вершин. Например, удобно передавать матрицы преобразований и проекций.

Внутри шейдера переменные uniform доступны только для чтения. Значения можно поменять только в основной программе для OpenGL. Однако в шейдере можно присвоить значение по умолчанию.

Переменные uniform можно объявлять сразу в нескольких шейдерах, но в этом случае тип переменной должен быть везде одинаковый.

Передадим, например, в шейдер матрицу поворота. Приведём программу вершинного шейдера:

```
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;
out vec3 Color;
uniform mat4 RotationMatrix;
void main() {
    Color = VertexColor;
    gl_Position = RotationMatrix * vec4(VertexPosition, 1.0);
}
```

Переменная `RotationMatrix` объявлена как uniform. Данные будут переданы из основной программы для OpenGL. `RotationMatrix` нужна для поворота вершины перед присвоением результата переменной `gl_Position`.

Фрагментный шейдер:

```
#version 400
in vec3 Color;
layout (location = 0) out vec4 FragColor;
void main() {
    FragColor = vec4(Color, 1.0);
}
```

Определим в основной программе матрицу вращения и пошлём её в шейдер. Для создания матрицы воспользуемся библиотекой GLM:

```
#include <glm/glm.hpp>
using glm::mat4;
using glm::vec3;
#include <glm/gtc/matrix_transform.hpp>
```

Не будем приводить оставшийся код, который должен скомпилировать и подключить шейдеры и нарисовать какой-нибудь примитив. Предположим, что `vaoHandle` — это указатель на массив вершин, а `programHandle` — это указатель на объект программы.

```
glClear(GL_COLOR_BUFFER_BIT);
mat4 rotationMatrix = glm::rotate(mat4(1.0f), angle, vec3(0.0f, 0.0f, 1.0f));
GLuint location = glGetUniformLocation(programHandle, "RotationMatrix");
if (location >= 0) {
    glUniformMatrix4fv(location, 1, GL_FALSE, &rotationMatrix[0][0]);
}
 glBindVertexArray(vaoHandle);
 glDrawArrays(GL_TRIANGLES, 0, 3);
```

Для присвоения значения переменной uniform нужно определить расположение переменной, а потом в найденное место записать значение, используя одну из `glUniform` функций.

Пример начинается очисткой буфера цвета и созданием с помощью GLM матрицы поворота. Далее, вызвав `glGetUniformLocation`, определяется расположение переменной uniform. Эта функция принимает указатель на объект шейдера и имя переменной

`uniform`, а возвращает её расположение. Если переменная `uniform` неактивна, возвращается `-1`.

Затем мы присваиваем значение переменной `uniform` с помощью `glUniformMatrix4fv`. Первый аргумент — это расположение переменной `uniform`. Второй — число передаваемых матриц (переменная `uniform` может быть массивом). Третья — булево значение, определяющее, нужно ли транспонировать матрицу при записи в переменную `uniform`. При работе с матрицами в GLM транспонирование не требуется, поэтому используем в этом случае `GL_FALSE`. Если матрица в массиве задана построчно, то необходимо `GL_TRUE`. Последний аргумент — это указатель на данные, передаваемые в переменную `uniform`.

Переменные `uniform` могут быть любого типа языка GLSL, включая массивы и структуры. В OpenGL есть функции вида `glUniform` с суффиксами для каждого типа данных. Например, чтобы присвоить переменной типа `vec3`, нужно использовать `glUniform3f` или `glUniform3fv`.

В случае массивов используются функции с `v` на конце. Кстати, при желании можно указать любое место в массиве с помощью оператора `[]`. Например, чтобы указать на место второго элемента массива `MyArray` мы запишем следующее:

```
GLuint location = glGetUniformLocation( programHandle, "MyArray[1]" );
```

В случае структур, её поля нужно инициализировать по отдельности. Как и в случае массива можно указать место конкретного поля, например:

```
GLuint location = glGetUniformLocation( programHandle, "MyMatrices.Rotation" );
```

Здесь переменная типа структура — это `MyMatrices`, а поле — `Rotation`.

1.0.8 Получение списка активных переменных `uniform`

1.0.9 Шейдеры (Shaders)

Типы векторов и матриц в GLSL

Базов. тип	2-мерн.	3-мерн.	4-мерн.	Матричные типы		
float	vec2	vec3	vec4	mat2	mat3	mat4
				mat2x2	mat2x3	mat2x4
				mat3x2	mat3x3	mat3x4
				mat4x2	mat4x3	mat4x4
double	dvec2	dvec3	dvec4	dmat2	dmat3	dmat4
				dmat2x2	dmat2x3	dmat2x4
				dmat3x2	dmat3x3	dmat3x4
				dmat4x2	dmat4x3	dmat4x4
int	ivec2	ivec3	ivec4	—	—	—
uint	uvec2	uvec3	uvec4	—	—	—
bool	bvec2	bvec3	bvec4	—	—	—

В матричных типах первое число — это количество колонок, второе — количество строк.

Инициализация переменных происходит аналогично скалярным

```
vec3 velocity = vec3(0.0, 2.0, 3.0);
```

Аналогично возможно преобразование типов:

```
ivec3 steps = ivec3(velocity);
```

Конструктор вектора можно использовать для укорачивания или удлинения вектора. Если передать более длинный вектор в конструктор более короткого, вектор обрежется до соответствующей длины.

В GLSL определены специальные типы переменных:

- **uniform** — связь шейдера с внешними данными, следует отметить, что этот тип переменных только для чтения;
- **varying** — этот тип переменных необходим для связи фрагментного шейдера с вершинным шейдером, то есть для передачи данных от вершинного шейдера к фрагментному. В вершинном шейдере они могут изменяться, а во фрагментном доступны только для чтения;
- **attribute** — переменные глобальной области видимости; Так же следует упомянуть о некоторых элементах языка GLSL, которые встречаются в примерах ниже:
- **sampler2D** — один из типов языка GLSL, представляющий текстуру (есть еще **sampler1D**, **sampler3D**, **samplerCube**, **sampler1Dshadow**, **sampler2Dshadow**);
- **vec4 texture2D(sampler2D s, vec2 coord)** — функция, используемая для чтения пикселя из текстуры s, с текстурными координатами coord.
- **gl_FrontColor** — это вектор, в который записываются конечные цветовые данные текстуры и который доступен только во фрагментном шейдере.

Глава 2

Текстуры в шейдерах

2.1 Двумерные текстуры

Наложение текстуры на поверхность в GLSL сводится к тому, что мы обращаемся к памяти, где расположена текстура, и получаем цвет, соответствующий текстурным координатам. Полученный цвет записываем в текущий пиксель на экране (во фрагментном шейдере). Конечный цвет пикселя на экране может быть взят напрямую из текстуры или быть полученным из смешения с цветом пикселя по модели Фонга. Доступ к текстурам в GLSL происходит посредством переменных *sampler*, которые работают как «указатели» на текстуры. В шейдере такие переменные обычно объявляют как *uniform*, а в основной программе OpenGL им присваивают указатель на текстуру.

Рассмотрим пример наложения двумерной текстуры на поверхность. Будем модифицировать цвет, полученный по Фонгу, цветом текстуры. На рис. 1.1 виден результат применения к кубу текстуры из кирпичей.

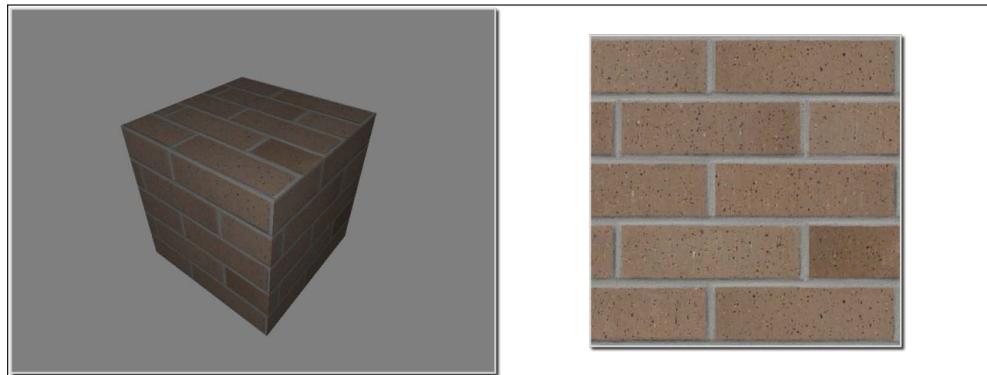


Рис. 1.1: Наложение двумерной текстуры.

Напишем программу OpenGL так, чтобы положение вершин имело номер атрибута 0, нормали 1, а текстурные координаты 2. Параметры модели Фонга передаются в шейдер как переменные *uniform* и должны быть заданными в основной программе OpenGL.

В следующем отрывке программы OpenGL в память компьютера загружается текстура. Используются библиотеки Qt, и считается, что *programHandle* — это указатель на шейдер.

```
// Загружаем файл с текстурой
const char * texName = "texture/brick1.jpg";
QImage img = QGLWidget::convertToGLFormat(QImage(texName, "JPG"));
// Передаём текстуру в OpenGL
glActiveTexture(GL_TEXTURE0);
GLuint tid;
 glGenTextures(1, &tid);
 glBindTexture(GL_TEXTURE_2D, tid);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, img.width(), img.height(), 0,
              GL_RGBA, GL_UNSIGNED_BYTE, img.bits());
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
// Присваиваем такое значение sampler-uniform-переменной Tex1,
// чтобы она ссылалась на нулевую текстуру
int loc = glGetUniformLocation(programHandle, "Tex1");
if( loc >= 0 )
    glUniform1i(loc, 0);
else
    fprintf(stderr, "Uniform-переменная Tex1 не обнаружена!\n");
```

Далее приведём текст вершинного шейдера:

```
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main() {
    TexCoord = VertexTexCoord;
    Normal = normalize(NormalMatrix * VertexNormal);
    Position = vec3(ModelViewMatrix *
        vec4(VertexPosition, 1.0));
    gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

Теперь текст фрагментного шейдера:

```
#version 400
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;
uniform sampler2D Tex1;
struct LightInfo {
    vec4 Position; // Положение источника света в системе координат наблюдателя.
    // Интенсивности света: A (внешнего), D (рассеянного) и S (зеркального)
    vec3 Intensity;
};
uniform LightInfo Light;
struct MaterialInfo {
    vec3 Ka; // Коэффициент отражения внешнего света
    vec3 Kd; // Коэффициент отражения рассеянного света
    vec3 Ks; // Коэффициент отражения зеркального света
    float Shininess; // Коэффициент зеркального блеска (Specular shininess factor)
};
uniform MaterialInfo Material;
layout(location = 0) out vec4 FragColor;
void phongModel( vec3 pos, vec3 norm, out vec3 ambAndDiff, out vec3 spec ) {
    // Вычисляем компоненты освещённости из модели Фонга. Внешнюю и
    // рассеянную освещённости помещаем в ambAndDiff,
    // а зеркальную - в spec.
```

```

    ...
}

void main() {
    vec3 ambAndDiff, spec;
    vec4 texColor = texture( Tex1, TexCoord ); // ◀ •••••

    phongModel(Position, Normal, ambAndDiff, spec);
    FragColor = vec4(ambAndDiff, 1.0) * texColor + vec4(spec, 1.0);
}

```

В первом отмыкке из программы OpenGL происходит (1) загрузка текстуры из файла, (2) передача её данных в OpenGL, и (3) присваивается значение переменной `sampler` для шейдера. Загрузка текстуры из файла сильно зависит от возможностей программной оболочки. Например, если подключить библиотеки Qt, то потребуются классы `QtGLWidget` и `QImage`. Конструктор класса `QImage` отвечает за загрузку файла. Первый аргумент — это путь к файлу, второй — тип картинки. Статический метод `convertToGLFormat` класса `QImage` преобразует картинку к формату OpenGL (формат `GL_RGBA`). Конечный результат храниться в переменной `timg` класса `QImage`.

Если Qt недоступна, то для загрузки картинки есть множество других способов, например: Devil (<http://openil.sourceforge.net/>), Freeimage (<http://freeimage.sourceforge.net/>) или SOIL (<http://www.lonesock.net/soil.html>).

Далее мы делаем текстуру с номером `GL_TEXTURE0` активной с помощью `glActiveTexture`. Создаём текстуру функцией `glGenTextures`. Указатель на текстуру хранится в переменной `tid`. С помощью `glBindTexture` задаём тип текстуры `GL_TEXTURE_2D`. Копируем пиксели текстуры в объект текстуры функцией `glTexImage2D` (последний аргумент — это указатель на пиксели картинки, в случае `QI` нужно использовать поле `bits` класса `QImage`).

В следующих шагах функцией `glTexParameterf` задаются увеличивающий и уменьшающий фильтры текстуры (в примере значение `GL_LINEAR`). Текстурный фильтр определяет, нужно применять интерполяцию для определения цвета текстуры в заданных координатах. Данный параметр оказывает существенной влияние на качество результата. В примере значение `GL_LINEAR` говорит о том, что будет возвращена взвешенная среднее четырёх пикселей ближайших к заданным координатам.

В конце мы присваиваем `uniform`-переменной `Tex1` ноль. Это `sampler`-переменная, она объявлена во фрагментном шейдере с типом `sampler2D`. Присваивая значение ноль, мы говорим OpenGL, что `Tex1` будет соответствовать текстуре `GL_TEXTURE0`.

В вершинном шейдере появилась новая переменная `TexCoord`. Она связана с атрибутом номер 2. Её значение передаётся во фрагментный шейдер через выходную (out) переменную `TexCoord`.

В фрагментном шейдере нам нужно задействовать переменную `Tex1`. Для доступа к текстуре существует встроенная функция `texture`. Первый аргумент — это `sampler`-переменная, определяющая текстуру, с которой будем сейчас работать. Второй аргумент — это текстурные координаты. Функция возвращает значение цвета в виде `vec4` (сохраняется в переменную `texColor`). Как было замечено выше, возвращаемый цвет — это интерполированное значение четырёх ближайших пикселей текстуры.

Результат вычислений по модели Фонга сохраняется в переменные `ambAndDiff` and

(только рассеянный и внешний свет) и `spec` (зеркальный свет). Как правило текстура взаимодействует только с рассеянным светом и не затрагивает зеркальную составляющую. Поэтому мы умножим цвет текстуры на внешнюю и рассеянную компоненты, а потом прибавим зеркальный свет. Полученный результат записывается в переменную `FragColor` во фрагментном шейдере.

2.2 Несколько текстур

Накладывая на объект сразу несколько текстур, можно получить различные интересные эффекты. Основная текстура может задавать «чистую» поверхность. Второй слой текстуры может давать дополнительные подробности, например: тень, дефекты поверхности, шероховатость или повреждение. Часто применяют так называемые световые карты, которые несут информацию об экспозиции (дозе светового облучения). Благодаря таким картам можно быстро рисовать тени и регулировать освещенность объектов, не прибегая явно к точному расчёту теней и отражений света. Такие текстуры называют «заранее испечённым (prebaked)» освещением.

Наложим для примера два слоя текстур. Основной слой будет полностью непрозрачным изображением кирпичей, а второй слой — частично прозрачным. Непрозрачные элементы второго слоя представляют собой мох, выросший на кирпичах (рис. 2.2).

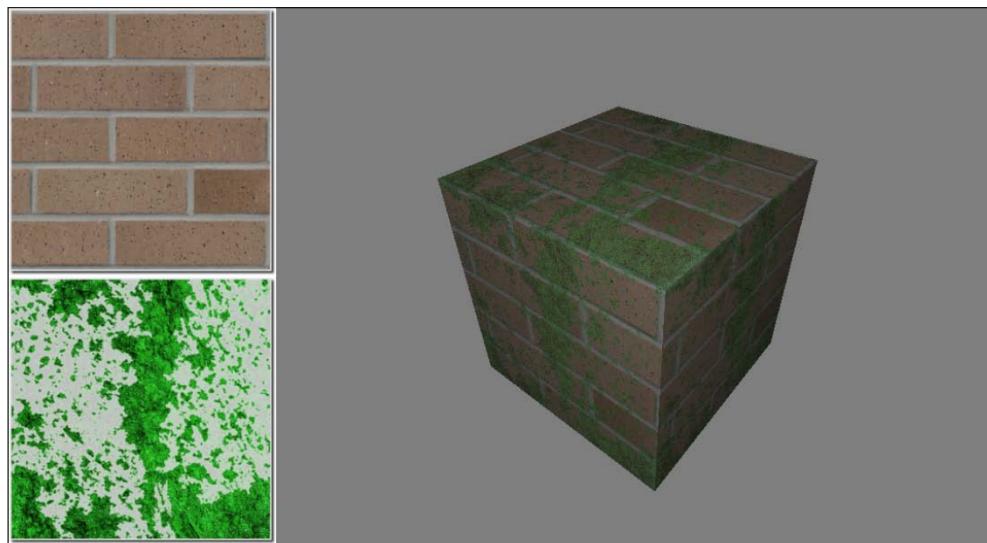


Рис. 2.2: Наложение двух текстур.

В основной программе OpenGL положения вершин будут хранится в атрибуте 0, нормали вершин — в атрибуте 1, а текстурные координаты — в атрибуте 2. Параметры для модели Фонга объявлены как переменные `uniform`, а их значение задаётся в основной программе OpenGL.

Во фрагментном шейдере есть две sampler-переменные: `BrickTex` и `MossTex`. Их нужно инициализировать, чтобы обращаться к соответствующим текстурам.

В программе OpenGL загружаем две текстуры в память: кирпичи имеют индекс 0, а мох — индекс 1.

```

GLuint texIDs[2];
glGenTextures(2, texIDs);
// Загружаем файл с текстурой из кирпичей
const char * texName = "texture/brick1.jpg";
QImage brickImg = QGLWidget::convertToGLFormat(QImage(texName, "JPG"));
// Переносим текстуру кирпичей в OpenGL
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texIDs[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, brickImg.width(),
brickImg.height(), 0, GL_RGBA, GL_UNSIGNED_BYTE,
brickImg.bits());
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
// Кирпичам (BrickTex) соответствует индекс 0
int uniloc = glGetUniformLocation(programHandle, "BrickTex");
if( uniloc >= 0 )
    glUniform1i(uniloc, 0);
// Загружаем файл с текстурой моха
texName = "texture/moss.png";
QImage mossImg =
QGLWidget::convertToGLFormat(QImage(texName, "PNG"));
// Переносим текстуру моха в OpenGL
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texIDs[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, mossImg.width(),
mossImg.height(), 0, GL_RGBA, GL_UNSIGNED_BYTE,
mossImg.bits());
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
// Mxy (MossTex) соответствует индекс 1
uniloc = glGetUniformLocation(programHandle, "MossTex");
if( uniloc >= 0 )
    glUniform1i(uniloc, 1);

```

Вершинный шейдер возьмём из предыдущего примера.

Во фрагментном шейдере поменяются sampler-переменная на:

```

uniform sampler2D BrickTex;
uniform sampler2D MossTex;

```

Основная функция фрагментного шейдера будет следующей:

```

void main() {
    vec3 ambAndDiff, spec;
    vec4 brickTextColor = texture( BrickTex, TexCoord );
    vec4 mossTextColor = texture( MossTex, TexCoord );
    phongModel(Position, Normal, ambAndDiff, spec);
    vec3 texColor = mix(brickTextColor, mossTextColor, mossTextColor.a);
    FragColor = vec4(ambAndDiff, 1.0) * texColor + vec4(spec, 1.0);
}

```

В основной программе OpenGL текстуры загружаются обычным образом. Внимания заслуживает только то, что им присваиваются различные индексы. Для кирпичей: `glActiveTexture(GL_TEXTURE0)`, для мха: `glActiveTexture(GL_TEXTURE1)`. Далее сразу инициализируем uniform-переменные: `BrickTex` получает значение 0, а `MossTex` — значение 1.

Во фрагментном шейдере мы извлекаем из каждой текстуры цвета в заданных текстурных координатах и сохраняем их в переменные: `brickTextColor` и `mossTextColor`. Два цвета смешиваются встроенной функцией `mix`. Третий параметр функции задаёт долевое соотношение смешиваемых цветов. α -компоненты (прозрачность) цвета

из текстуры мха определяет пропорции цветов при смешивании. Результат смешивания — это линейная интерполяция двух цветов в точке, определяемой α -компонентой мха. В OpenGL есть похожая функция смешивания: `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`, причём `moх` — это цвет «источника», а кирпичи — это цвет «приёмника».

В конце, как и раньше, умножаем результат функции `mix` на внешний и рассеянный свет, а потом прибавляем зеркальную компоненту.

В примере при смешивании цветов была использована α -компонента второй текстуры. Но это лишь один из возможных вариантов взаимодействия текстур.

Распространённый подход состоит в использовании дополнительного атрибута вершины, где указано, как смешивать текстуры. Получается, что в разных вершинах текстуры можно смешивать в разных пропорциях. Например, мы могли бы менять количество мха на поверхности куба, определив ещё один атрибут вершины, где указана пропорция при смешивании кирпичей и мха. Нулевое значение говорит об отсутствии мха, а при единице используется только α -компонента мха.

2.3 α -карты для создания дырок и отверстий

Чтобы создать объект, имеющий отверстия, можно использовать текстуры с α -каналом, где есть информация о прозрачных частях объекта. При этом нужно настроить буфер глубины только на чтение и отрисовывать все наши полигоны от самого дальнего до ближнего, чтобы избежать неверного наслаждения (перемешивания) полигонов друг на друга. Нам понадобилось бы отсортировать все полигоны по расстоянию до наблюдателя и отрисовать их в правильном порядке. Это сложная задача.

На помощь приходят шейдеры. В GLSL есть ключевое слово `discard`. С его помощью шейдер может пропускать пиксели на экране, если значение α в текстуре меньше определённого значения. Если полностью пропускать обработку пикселей экрана, не нужно уже менять буфер глубины, так как эти пиксели не повлияют на буфер глубины. Отпадает необходимость сортировать полигоны по глубине, так как нет смешивания (blending).

На рис. 3.3 справа изображён чайник с дырявыми стенками. Расположение дыр определяется текстурой слева. Фрагментный шейдер пропускает пиксели экрана, которым соответствует цвет текстуры со значением α ниже порогового значения.

Дырки позволяют наблюдателю заглянуть внутрь объекта. Поэтому сквозь них могут быть видны некоторые внутренние стены. В связи с этим нужно включить двустороннее освещение поверхностей (чтобы видеть и лицевые, и задние грани).

Загрузим текстуру в основной программе OpenGL. Сопоставим ей индекс 0, а α -карте — индекс 1. Присвоим ноль uniform-переменной `BaseTex` и единицу — переменной `AlphaTex`.

1. Возьмём тексты обоих шейдеров из примера с двумя текстурами. Во фрагментном шейдере сделаем следующие изменения. Заменим имеющиеся uniform-переменные на

```
uniform sampler2D BaseTex;
uniform sampler2D AlphaTex;
```

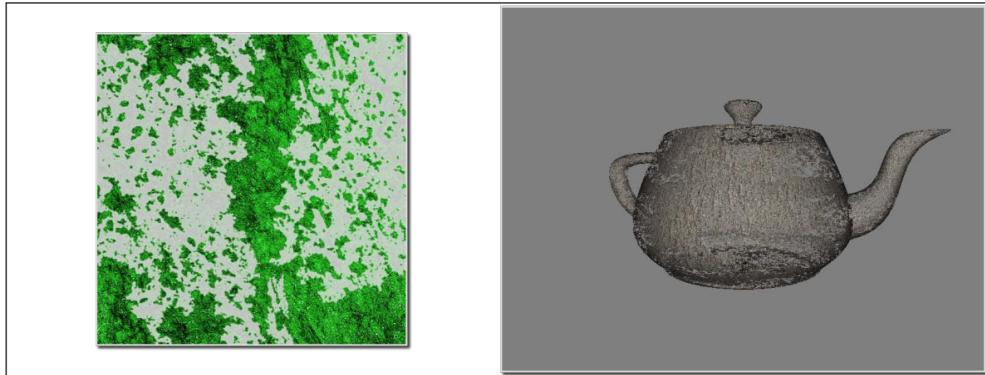


Рис. 3.3: Дырки в снетках объекта определяются α -компонентом цвета текстуры.

Основную функцию заменим следующей

```
void main() {
    vec4 baseColor = texture( BaseTex, TexCoord );
    vec4 alphaMap = texture( AlphaTex, TexCoord );
    if(alphaMap.a < 0.15 )
        discard;
    else {
        if( gl_FrontFacing ) {
            FragColor = vec4(phongModel(Position,Normal),1.0) * baseColor;
        } else {
            FragColor = vec4(phongModel(Position,-Normal),1.0) * baseColor;
        }
    }
}
```

В функции `main` фрагментного шейдера мы получаем цвет пикселя из текстуры и сохраняем его в `baseColor`. Значение из текстуры с α -картой записываем в `alphaMap`. Если значение `alphaMap` меньше фиксированного значения (0,15 в примере), текущий пиксель экрана не обрабатывается шейдером из-за присутствия ключевого слова `discard`.

В противном случае вычисляем освещённость по Фонгу. Нужно обратить внимание на направление вектора нормали, так как грани могут быть лицевые и задние. Освещённость из модели Фонга умножается на цвет из `BaseTex`.

Описанный подход довольно простой и ясный. Он является хорошей альтернативой традиционным техникам смешивания. Метод позволяет хорошо имитировать дыры в объектах или следы разрушения. В случае, когда значения α -карты меняются непрерывно по площади карты, можно создать эффект динамического разрушения объекта во времени. Можно менять пороговое значение α от 0,0 до 1,0, при этом объект будет постепенно разрушаться до полного исчезновения.

2.4 Карты нормалей

На рис. 4.4 изображена сетка чудища с использованием карты нормалей и без неё. Вверху слева основная текстура чудища. В нашем примере текстура влияет на рассеянную компоненту света. Вверху справа чудище с цветной текстурой и обычными нормалями. Внизу слева карта нормалей. Внизу справа чудище с цветной текстурой

и картой нормалей. Дополнительные морщины на лице возникают благодаря карте нормалей.

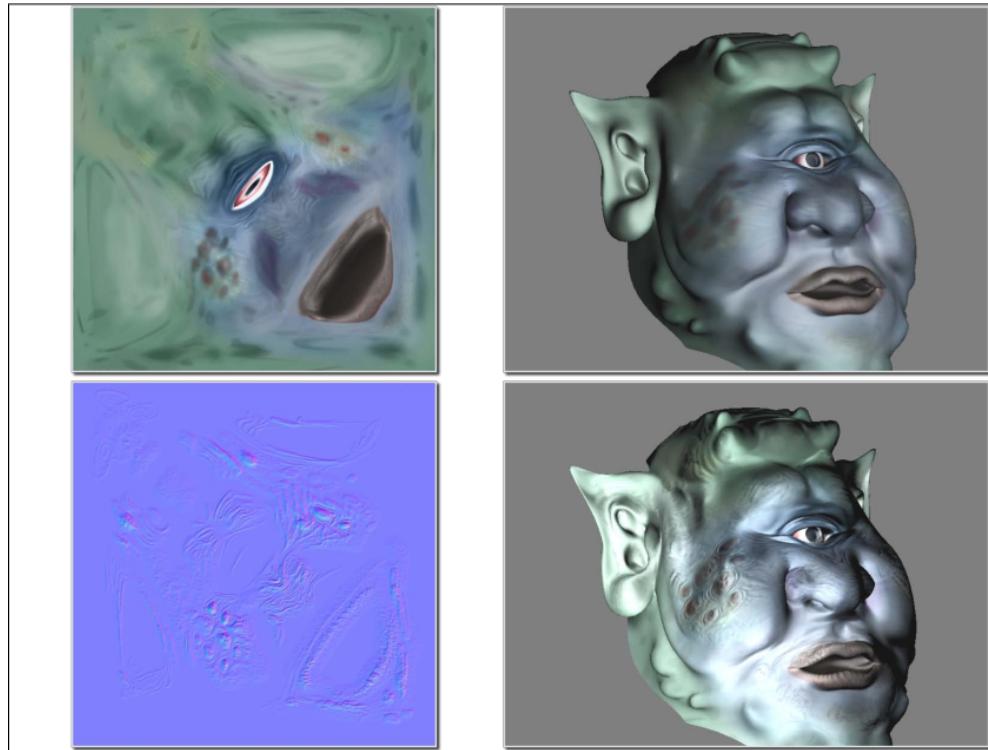


Рис. 4.4: Сетка чудища с использованием карты нормалей и без неё. Вверху слева основная текстура чудища. В нашем примере текстура влияет на рассеянную компоненту света. Вверху справа чудище с цветной текстурой и обычными нормалями. Внизу слева карта нормалей. Внизу справа чудище с цветной текстурой и картой нормалей.

Карту нормалей можно получить многими способами. Многие программы 3D-моделирования (например, Maya, Blender или 3D Studio Max) могут создавать карты нормалей. Карта нормалей может быть получена из карты высот (карту высот можно рассматривать как изображение в градациях серого). Существует plugin от NVIDIA для Adobe Photoshop, позволяющий это сделать (<https://developer.nvidia.com/nvidia-texture-tools>).

1. Use the following code for the vertex shader:

```
#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

layout (location = 2) in vec2 VertexTexCoord;
layout (location = 3) in vec4 VertexTangent;
struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // A,D,S intensity
};
uniform LightInfo Light;
out vec3 LightDir;
out vec2 TexCoord;
out vec3 ViewDir;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
```

```

uniform mat4 MVP;
void main() {
    // Transform normal and tangent to eye space
    vec3 norm = normalize(NormalMatrix * VertexNormal);
    vec3 tang = normalize(NormalMatrix * vec3(VertexTangent));
    // Compute the binormal
    vec3 binormal = normalize( cross( norm, tang ) ) *
        VertexTangent.w;
    // Matrix for transformation to tangent space
    mat3 toObjectLocal = mat3(
        tang.x, binormal.x, norm.x,
        tang.y, binormal.y, norm.y,
        tang.z, binormal.z, norm.z );
    // Get the position in eye coordinates
    vec3 pos = vec3( ModelViewMatrix *
        vec4(VertexPosition,1.0) );
    // Transform light dir. and view dir. to tangent space
    LightDir = normalize( toObjectLocal *
        (Light.Position.xyz - pos) );
    ViewDir = toObjectLocal * normalize(-pos);
    // Pass along the texture coordinate
    TexCoord = VertexTexCoord;
    gl_Position = MVP * vec4(VertexPosition,1.0);
}

```

2. Use the following code for the fragment shader:

```

#version 400
in vec3 LightDir;
in vec2 TexCoord;
in vec3 ViewDir;
uniform sampler2D ColorTex;
uniform sampler2D NormalMapTex;
struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // A,D,S intensity
};
uniform LightInfo Light;
struct MaterialInfo {
    vec3 Ka; // Ambient reflectivity
    vec3 Ks; // Specular reflectivity
    float Shininess; // Specular shininess factor
};
uniform MaterialInfo Material;
layout( location = 0 ) out vec4 FragColor;
vec3 phongModel( vec3 norm, vec3 diffR ) {
    vec3 r = reflect( -LightDir, norm );
    vec3 ambient = Light.Intensity * Material.Ka;
    float sDotN = max( dot(LightDir, norm), 0.0 );
    vec3 diffuse = Light.Intensity * diffR * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Intensity * Material.Ks *
            pow( max( dot(r, ViewDir), 0.0 ),
                Material.Shininess );
    return ambient + diffuse + spec;
}

void main() {
    // Lookup the normal from the normal map
    vec4 normal = texture( NormalMapTex, TexCoord );
    // The color texture is used as the diffuse reflectivity
    vec4 texColor = texture( ColorTex, TexCoord );
    FragColor = vec4( phongModel(normal.xyz, texColor.rgb),
        1.0 );
}

```

2.5 Имитация зеркального отражения с помощью кубических карт

1. Load the six images of the cube map into a single texture target using the following code within the main OpenGL program:

```

glActiveTexture(GL_TEXTURE0);
GLuint texID;
 glGenTextures(1, &texID);
 glBindTexture(GL_TEXTURE_CUBE_MAP, texID);
 const char * suffixes[] = { "posx", "negx", "posy",
                             "negy", "posz", "negz" };
 GLuint targets[] = {
    GL_TEXTURE_CUBE_MAP_POSITIVE_X,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
    GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
    GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
};

for( int i = 0; i < 6; i++ ) {
    string texName = string(baseFileName) + "_" + suffixes[i] + ".png";
    QImage img = QGLWidget::convertToGLFormat(QImage(texName.c_str(),"PNG"));
    glTexImage2D(targets[i], 0, GL_RGBA, img.width(), img.height(),
                 0, GL_RGBA, GL_UNSIGNED_BYTE, img.bits());
}

// Typical cube map settings
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
// Set the CubeMapTex uniform to texture unit 0
uniloc = glGetUniformLocation(programHandle, "CubeMapTex");
if( uniloc >= 0 )
    glUniform1i(uniloc, 0);

```

2. Use the following code for the vertex shader:

```

#version 400
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
out vec3 ReflectDir; // The direction of the reflected ray
uniform bool DrawSkyBox; // Are we drawing the sky box?
uniform vec3 WorldCameraPosition;
uniform mat4 ModelViewMatrix;
uniform mat4 ModelMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void main()
{
    if( DrawSkyBox ) {
        ReflectDir = VertexPosition;
    } else {
        // Compute the reflected direction in world coords.
        vec3 worldPos = vec3( ModelMatrix *
            vec4(VertexPosition, 1.0) );
        vec3 worldNorm = vec3(ModelMatrix *
            vec4(VertexNormal, 0.0));
        vec3 worldView = normalize( WorldCameraPosition -
            worldPos );
        ReflectDir = reflect(-worldView, worldNorm );
    }
}

```

```
        }
        gl_Position = MVP * vec4(VertexPosition, 1.0);
    }
```

3. Use the following code for the fragment shader:

```
#version 400
in vec3 ReflectDir; // The direction of the reflected ray
uniform samplerCube CubeMapTex; // The cube map
uniform bool DrawSkyBox; // Are we drawing the sky box?
uniform float ReflectFactor; // Amount of reflection
uniform vec4 MaterialColor; // Color of the object's "Tint"
layout( location = 0 ) out vec4 FragColor;
void main() {
    // Access the cube map texture
    vec4 cubeMapColor = texture(CubeMapTex, ReflectDir);
    if( DrawSkyBox )
        FragColor = cubeMapColor;
    else
        FragColor = mix(MaterialColor, CubeMapColor, ReflectFactor);
}
```

4. In the render portion of the OpenGL program, set the uniform DrawSkyBox to true, and then draw a cube surrounding the entire scene, centered at the origin. This will become the sky box. Following that, set DrawSkyBox to false, and draw the object(s) within the scene.

Глава 3

Мозаичные шейдеры

3.1 Геометрические и мозаичные шейдеры

Геометрические и мозаичные шейдеры появились в OpenGL недавно.

Геометрические шейдеры используются для добавления, изменения или удаления геометрии. Мозаичные шейдеры нужны для автоматического создания геометрии, основанной на интерполяции входных данных (опорных точек).

Рассмотрим несколько примеров геометрических и мозаичных шейдеров.

На рис. 1.1 приведена последовательность вызова шейдеров.

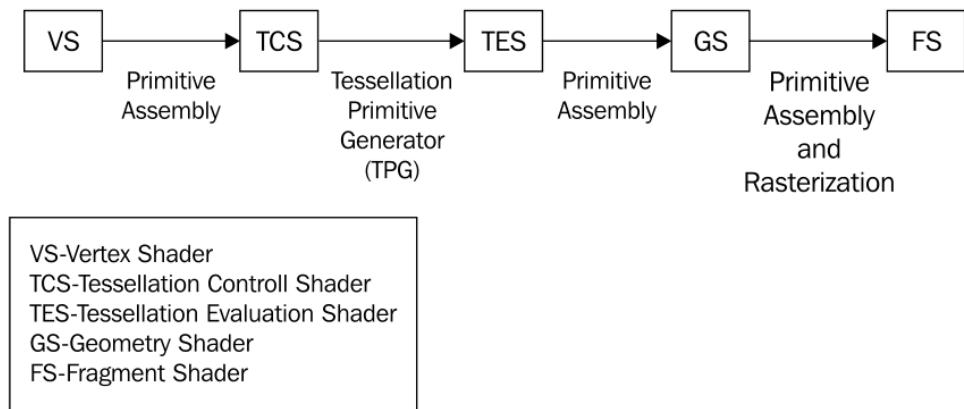


Рис. 1.1: Последовательность вызова шейдеров. VS — Vertex Shader (вершинный шейдер), TCS — Tessellation Control Shader (мозаичный управляющий шейдер), TES — Tessellation Evaluation Shader (мозаичный вычисляющий шейдер), GS — Geometry Shader (геометрический шейдер), FS — Fragment Shader (фрагментный шейдер).

3.2 Геометрический шейдер

Геометрический шейдер (geometry shader, GS) вызывается один раз для каждого примитива. Он может работать сразу со всеми вершинами примитива, включая все входные переменные, связанные с вершинами. Другими словами, если на предыдущем шаге (в вершинном шейдере) задаются выходные переменные, геометрический шейдер

может считывать эти значения для всех вершин примитива. Очевидно, все входные переменные в геометрическом шейдере всегда являются массивами.

Геометрический шейдер может выдавать ноль, один или несколько примитивов. Примитивы на выходе могут идти тип, отличный от типа примитива на входе. Однако ГШ не может создавать примитивы различных типов. Например, ГШ может принимать на входе треугольник, а выдавать сегменты ломанной (line strip). Или ГШ может принимать треугольник, а выдавать ноль или несколько треугольников в виде треугольной сетки (triangle strip).

Всё это позволяет ГШ находить применение в различных задачах. Приведём далее несколько иллюстраций. ГШ можно использовать для удаления части примитивов с помощью некоторого критерия, например, когда некоторый объект заслоняет часть примитивов. ГШ может добавлять примитивы, чтобы расширить форму отображаемого объекта. ГШ может только рассчитать дополнительную информацию о примитиве и передать примитив неизменным. Наконец ГШ может создавать совершенно новые примитивы по сравнению с теми, что были на входе.

Работа ГШ всегда сосредоточена вокруг двух встроенных функций: `EmitVertex` и `EndPrimitive`. Эти две функции позволяют ГШ посыпать в графический конвейер несколько вершин и примитивов. ГШ определяет выходные переменные для конкретной вершины и затем вызывает `EmitVertex`. После этого ГШ может продолжить работу и переопределить выходные переменные уже для другой вершины. После этого снова вызывается `EmitVertex` и так далее. Когда все вершины примитива заданы, ГШ вызывает `EndPrimitive`, чтобы сообщить OpenGL о готовности всех вершин примитива. Функция `EndPrimitive` неявно вызывается, когда ГШ заканчивает свою работу. Если ГШ ни разу не вызывает `EmitVertex`, то примитив на входе фактически не проходит дальше по графическому конвейеру (не отображается).

Далее мы рассмотрим некоторые примеры геометрических шейдеров. В «точечных рисунках» тип примитива на входе ГШ отличается от типа на выходе. В «проводочном каркасе поверх объекта» примитивы передаются в неизменном виде, но внутри ГШ вычисляются дополнительные данные, необходимые для конечной отрисовки. В «силуэте» ГШ передаёт исходные примитивы, добавляя к ним ещё свои собственные.

3.3 Разбиение кривой

Рассмотрим в качестве простого примера построение кубической кривой Безье по 4 опорным точкам:

$$P(t) = B_0^3(u)\mathbf{p}_0 + B_1^3(u)\mathbf{p}_1 + B_2^3(u)\mathbf{p}_2 + B_3^3(u)\mathbf{p}_3, \quad 0 \leq u \leq 1,$$

$$B_0^3(u) = (1-u)^3, \quad B_1^3(u) = 3(1-u)^2u, \quad B_2^3(u) = 3(1-u)u^2, \quad B_3^3(u) = u^3.$$

Тесселяция состоит из двух фаз: настройка тесселяции (TCS) и расчёт тесселяции (TES). Кривая Безье будет приближена ломаной. Количество отрезков ломаной задаётся в TCS (уровень наружной тесселяции). Координаты вершин ломаной вычисляются в TES. На рис. 3.2 приведены три различных уровня тесселяции.

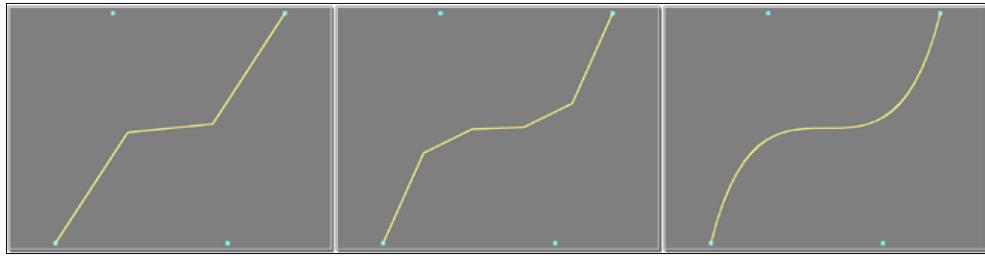


Рис. 3.2: Различные уровни тесселяции кривой Безье. Слева уровень 3, в центре уровень 5, справа уровень 30. Опорные точки кривой Безье обозначены маленькими квадратиками.

Опорные точки кривой Безье передаются в шейдерный конвейер как примитив типа `patch`, состоящий из 4 вершин. Примитивы этого типа задаются программистом. Они состоят из вершин, которые можно использовать на усмотрение программиста. TCS исполняется один раз для каждой вершины примитива. Количество вызова TES не фиксировано, оно задаётся числом вершин на выходе TPG. В результате тесселяции получается набор примитивов. В нашем случае таким примитивом будет ломаная, приближающая кривую Безье.

Внутри TCS определяется уровень тесселяции. Очень грубо уровень тесселяции можно понимать как количество порождённых вершин. В нашем случае формируется ломаная, поэтому уровень тесселяции совпадает с числом отрезков ломаной. Каждая порождённая вершина будет иметь мозаичную координату (от 0 до 1). Обозначим её через u .

Если быть более точным, то внутри TCS включается механизм создания множества (более одной) ломаных, которые называются *изолиниями*. Для каждой изолинии координата u меняется от 0 до 1, а координата v постоянна. Количество различных значений u и v соответствуют двум уровням наружной тесселяции. В нашем примере создается только одна ломаная, поэтому вторая координата v всегда будет равна 1.

Основная задача внутри TES — определить положение вершины, связанной в текущем выполнением шейдера. У нас есть доступ к координатам u и v , связанным с вершиной. Также мы имеем доступ (только чтение) ко всем вершинам примитива. Мы можем определить положение вершины, используя приведённое выше параметрическое уравнение с параметром u .

Рассмотрим важные uniform-переменные для следующего примера:

- `NumSegments` Число получаемых разбиений.
- `NumStrips` Число получаемых изолиний. В примере это 1.
- `LineColor` Цвет ломаной.

Зададим в основной программе OpenGL переменные `uniform`. Всего будет скомпилировано и собрано 4 шейдера: вершинный, фрагментный, мозаичный управляющий и мозаичный вычисляющий.

Простейший вершинный шейдер:

```
#version 400
layout (location = 0 ) in vec2 VertexPosition;
void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

Управляющий мозаичный шейдер:

```
#version 400
layout( vertices=4 ) out;
uniform int NumSegments;
uniform int NumStrips;
void main()
{
    // Оставляем положения вершин неизменными
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
    // Определяем уровни разбиений (это работает для драйвера ATI Catalyst, для
    // прочих драйверов может потребоваться поменять местами присваиваемые значения)
    gl_TessLevelOuter[0] = float(NumSegments);
    gl_TessLevelOuter[1] = float(NumStrips);
}
```

Мозаичный вычислитель:

```
#version 400
layout( isolines ) in;
uniform mat4 MVP; // projection * view * model
void main()
{
    // Координата разбиения u
    float u = gl_TessCoord.x;
    // Вершины примитива (опорные точки)
    vec3 p0 = gl_in[0].gl_Position.xyz;
    vec3 p1 = gl_in[1].gl_Position.xyz;
    vec3 p2 = gl_in[2].gl_Position.xyz;
    vec3 p3 = gl_in[3].gl_Position.xyz;
    float u1 = (1.0 - u);
    float u2 = u * u;
    // Многочлены Бернштейна в точке u
    float b3 = u2 * u;
    float b2 = 3.0 * u2 * u1;
    float b1 = 3.0 * u * u1 * u1;
    float b0 = u1 * u1 * u1;
    // Кубическая интерполяция Безье
    vec3 p = p0 * b0 + p1 * b1 + p2 * b2 + p3 * b3;
    gl_Position = MVP * vec4(p, 1.0);
}
```

Фрагментный шейдер:

```
#version 400
uniform vec4 LineColor;
layout ( location = 0 ) out vec4 FragColor;
void main()
{
    FragColor = LineColor;
}
```

Важно задать число вершин в примитиве внутри приложения OpenGL. Это можно следить с помощью функции `glPatchParameter`:

```
glPatchParameteri( GL_PATCH_VERTICES, 4);
```

Зададим примитив типа `GL_PATCHES` в основной программе OpenGL:

```
glDrawArrays(GL_PATCHES, 0, 4);
```

Вершинный шейдер не несёт полезной нагрузки. Он передаёт положение вершины на следующий шаг без изменений.

Управляющий мозаичный шейдер начинается с задания числа вершин в выходном примитиве: `layout (vertices = 4) out;`

Заметим, что это не число генерируемых мозаичным шейдером вершин. В данном случае примитив состоит из 4 опорных точек, поэтому мы используем значение 4.

В основной функции TCS входные положения вершин (примитива) передаются в выходные положения без изменений. Массивы `gl_out` и `gl_in` содержать входную и выходную информацию о вершинах примитива. Отметим, что мы используем один и тот же индекс `gl_InvocationID` при работе с обоими массивами. Переменная `gl_InvocationID` определяет номер вершины выходного примитива, который обрабатывается текущим вызовом TCS. Шейдер TCS имеет доступ ко всем индексам массива `gl_in`, но может записывать в `gl_out` только в индекс `gl_InvocationID`.

Далее TSC устанавливает уровни разбиения в массиве `gl_TessLevelOuter`. Отметим, что присваиваемое число нецелое. Оно автоматически округляется системой OpenGL вверх до ближайшего целого и обрезается (при необходимости) для допустимого диапазона значений (если число больше максимально возможного, то используется максимально возможное).

Первый элемент в массиве определяет число участков разбиения. Каждой вершине ломаной соответствует некоторое значение параметра *u* от 0 до 1. Второй элемент массива — это число генерируемых изолиний. Каждой изолинии соответствует постоянное значение *v*. В нашем примере значение `gl_TessLevelOuter[1]` должно ровняться единице. Значение индексов массива зависит от используемого драйвера графической подсистемы.

Существовала неоднозначность при работе с `gl_TessLevelOuter` в случае изолиний. Для драйверов ATI Catalyst `gl_TessLevelOuter[0]` задаёт число сегментов в каждой изолинии. Но для драйвера NVIDIA нужно использовать `gl_TessLevelOuter[1]`. Ошибка, вероятно, будет скоро исправлена.

Мозаичный вычислитель (TES) начинается заданием типа примитива:

`layout (isolines) in;`

то определяет алгоритм разбиения, по которому будет работать мозаичный генератор примитивов. Здесь ещё допустимы значения `quads` и `triangles`.

Внутри основной функции TES переменная `gl_TessCoord` содержит координаты *u* и *v* для данного вызова TES. Мы осуществляляем разбиения только в одном направлении, поэтому будем использовать координату *u*, соответствующую координате *x* в `gl_TessCoord`.

В следующем шаге обращаемся к 4 опорным точкам (все точки нашего примитива). Они доступны в массиве `gl_in`.

Затем кубические многочлены Бернштейна вычисляются в точке *u* и запоминаются в `b0`, `b1`, `b2` и `b3`. Как было описано выше мы получаем точку кривой Безье при значении параметра *u*. Окончательное положение приводится к координатам экрана и присваивается переменной `gl_Position`.

Во фрагментном шейдере лишь задаётся цвет `LineColor`.

В следующем примере мы рассмотрим двумерное разбиение.

3.4 Тесселяция (замощение) плоского четырёхугольника

Один из лучших способов понять механизм тесселяции OpenGL – пронаблюдать тесселяцию плоского четырёхугольника. При использовании линейной интерполяции создаваемые треугольники непосредственно связаны с координатами тесселяции (u, v) на выходе генератора примитивов (TPG). Будет очень полезно нарисовать несколько четырёхугольников с разными внешними и внутренними уровнями разбиения и рассмотреть получившиеся треугольники. Именно это мы и сделаем.

При тесселяции четырёхугольников генератор примитивов разбивает пространство параметров (u, v) на некоторое число областей, зависящее от шести параметров, а именно от внутренних уровней разбиения для u и v (внешний уровень 0 и внутренний уровень 1) и внешних уровней разбиения для u и v по сторонам (внешние уровни 0 – 3). Так задаётся число разбиений по внешним границам пространства параметров и внутри него. Давайте рассмотрим каждый параметр отдельно: внешний уровень 0 (ВнешУ0) – число разбиений по v при $u = 0$; внешний уровень 1 (ВнешУ1) – число разбиений по u при $v = 0$; внешний уровень 2 (ВнешУ2) – число разбиений по v при $u = 1$; внешний уровень 3 (ВнешУ3) – число разбиений по u при $v = 1$; внутренний уровень 0 (ВнутУ0) – число разбиений по u при любых v внутри пространства параметров; внутренний уровень 1 (ВнутУ1) – число разбиений по v при любых u внутри пространства параметров;

На приведённом ниже рисунке показано, как связаны уровни разбиения с теми областями пространства тесселяции, на которые каждый из них влияет. Внешние уровни разбиения отвечают за число разбиений по сторонам, а внутренние – за внутреннее разбиение.

Описанные выше шесть уровней разбиения могут быть заданы массивами `gl_TessLevelOuter` и `gl_TessLevelInner`. Например, `gl_TessLevelInner[0]` соответствует ВнешУ0, `gl_TessLevelOuter[2]` – ВнешУ2 и т.д.

Если мы нарисуем примитив типа `patch`, который состоит из единственного четырёхугольника (четыре вершины), и используем линейную интерполяцию, то получившиеся треугольники помогут нам понять, как OpenGL реализует тесселяцию четырёхугольников. На приведённом ниже рисунке представлены результаты тесселяции для разных уровней разбиения.

Когда мы используем линейную интерполяцию, получившиеся треугольники представляют собой визуальное отображение пространства параметров (u, v). По оси x отложена координата u , а по оси y – координата v . Вершины треугольников – это координаты, полученные на выходе генератора примитивов. Число разбиений легко определить, посмотрев на получившуюся сетку из треугольников. Например, когда внешние уровни равны 2, а внутренние составляют 8, можно видеть, что внешние стороны имеют по 2 разбиения, а внутри четырёхугольника u и v разделены на 8 промежутков.

Так в оригиналe. Если с внешними уровнями всё очевидно, то объяснение товарища Вольфа относительно внутренних уровней мне не понятно. Внутри квадрата 2-8 по его описанию я могу насчитать для разных сечений и 10, и даже 20 промежутков. Исходя из рисунков, я бы сформулировала закономерность так: Любая прямая, параллельная одной из осей и проведённая через одну или более вершин треугольников (причём хотя бы одна из вершин должна лежать не на сторонах замощаемой фигуры), разбивается этими вершинами на число промежутков, равное значению внутреннего уровня разбиения.

Перед тем, как обратиться к коду, давайте поговорим о линейной интерполяции. Если углы четырёхугольника расположены так, как показано на рисунке ниже, то любую точку внутри фигуры можно задать путём линейной интерполяции вершин с учётом параметров u и v .

Мы запустим генератор примитивов, который создаст набор вершин с соответствующими параметрическими координатами, а положение этих точек мы определим путём интерполяции вершин четырёхугольника по приведённой выше формуле.

Подготовка

Внешние и внутренние уровни разбиения будут заданы переменными типа `uniform`: `Inner` и `Outer`. Чтобы отобразить треугольники, мы используем геометрический шейдер, описанный ранее в этой главе.

Создайте приложение OpenGL, которое отображает примитив типа `patch`, состоящий из четырёх вершин, расположенных, как показано на рисунке выше, в порядке против часовой стрелки.

Как это сделать...

Чтобы создать программу шейдера, которая будет генерировать набор треугольников для тесселяции четырёхугольника, заданного вершинами, следуйте такому алгоритму: Используйте следующий вершинный шейдер:

```
#version 400
layout (location = 0 ) in vec2 VertexPosition;
void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

Используйте следующий код в качестве мозаичного управляющего шейдера:

```
#version 400
layout( vertices=4 ) out;
uniform int Outer;
uniform int Inner;
void main()
{
    // Передаём исходное положение вершин без изменения
    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
    gl_TessLevelOuter[0] = float(Outer);
    gl_TessLevelOuter[1] = float(Outer);
    gl_TessLevelOuter[2] = float(Outer);
    gl_TessLevelOuter[3] = float(Outer);
    gl_TessLevelInner[0] = float(Inner);
    gl_TessLevelInner[1] = float(Inner);
}
```

Используйте следующий код в качестве мозаичного вычисляющего шейдера:

```
#version 400
layout( quads, equal_spacing, ccw ) in;
uniform mat4 MVP;
void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    vec4 p0 = gl_in[0].gl_Position;
    vec4 p1 = gl_in[1].gl_Position;
    vec4 p2 = gl_in[2].gl_Position;
    vec4 p3 = gl_in[3].gl_Position;
    // Линейная интерполяция
    gl_Position =
        p0 * (1-u) * (1-v) +
        p1 * u * (1-v) +
        p3 * v * (1-u) +
        p2 * u * v;
    // Переход к однородным координатам
    gl_Position = MVP * gl_Position;
}
```

Используйте геометрический шейдер из примера «Рисование проволочного каркаса поверх затенённой сетки».

Используйте следующий код в качестве фрагментного шейдера:

```
#version 400
uniform float LineWidth;
uniform vec4 LineColor;
uniform vec4 QuadColor;
noperspective in vec3 EdgeDistance; // Из геометрического шейдера
layout ( location = 0 ) out vec4 FragColor;
float edgeMix()
{
    // ** Сюда вставьте код, определяющий интенсивность
    // цвета кромки (см. пример «Рисование проволочного каркаса
    // поверх затенённой сетки»). **
}
void main()
{
    float mixVal = edgeMix();
    FragColor = mix( QuadColor, LineColor, mixVal );
}
```

Внутри функции визуализации вашей основной программы OpenGL задайте число вершин примитива: `glPatchParameteri(GL_PATCH_VERTICES, 4);`

Постройте изображение примитива в виде четырёх вершин на плоскости в порядке против часовой стрелки.

Как это работает...

Вершинный шейдер передаёт положение вершины мозаичному управляющему шейдеру без изменений.

В мозаичном управляющем шейдере число вершин задаётся следующей командой: `layout (vertices=4) out;`

В функции `main` мозаичного управляющего шейдера не подвергается изменениям положение вершины, но задаются внешние и внутренние уровни разбиения. Всем четырём внешним уровням присваивается значение Outer, а обоим внутренним – значение Inner. В мозаичном вычислителе мы задаём тип разбиения и его параметры: `layout (quads, equal_spacing, ccw) in;`

Параметр `quads` означает, что пространство параметров будет разбиваться генератором примитивов как четырёхугольник, как это было описано выше. Параметр `equal_spacing` говорит о том, что все получившиеся при разбиении промежутки должны иметь одинаковую длину. Последний параметр, `ccw`, означает, что примитивы должны создаваться в порядке против часовой стрелки.

Функция `main` мозаичного вычислителя начинается с получения параметрических координат данной вершины из переменной `gl_TessCoord`. Далее мы переходим к считыванию положений четырёх вершин из массива `gl_in`. Мы будем хранить их во временных переменных для расчёта интерполяции.

Затем во встроенную выходную переменную `gl_Position` записывается вычисленное по приведённой выше формуле значение интерполированной точки. И в конце мы переходим к однородным координатам путём умножения на матрицу проекции и трансформации модели.

Во фрагментном шейдере мы присваиваем всем фрагментам цвет, который, возможно, будет смешан с цветом контура, чтобы выделить кромки.

См. также

[Рисование проволочного каркаса поверх затенённой сетки](#)

[Тесселяция объёмной поверхности](#)

В качестве примера тесселяции объёмной поверхности давайте снова рассмотрим «чайникаэдр». Оказывается, что описывающие его данные представляют собой наборы из 16 контрольных точек, подходящие для интерполяции кубическими поверхностями Безье. Поэтому рисование чайника сводится к построению кубических поверхностей Безье.

Конечно, это прекрасная постановка задачи для мозаичных шейдеров. Каждый элемент поверхности, описываемый шестнадцатью вершинами, мы будем обрабатывать как примитив типа `patch`, разбъём пространство параметров как четырёхугольник и применим интерполяцию поверхностями Безье в мозаичном вычисляющем шейдере.

На рисунке ниже можно увидеть, чего мы хотим добиться. Чайник слева построен с внутренним и внешним уровнем разбиения 2, для среднего был задан уровень 4, а для правого – уровень 16. Мозаичный вычисляющий шейдер производит расчёты для интерполяции поверхностями Безье.

Для начала давайте рассмотрим, как происходит интерполяция кубическими поверхностями Безье. Если наша поверхность определяется набором из 16 контрольных точек (размещённых в сетке 4×4) P_{ij} , где i и j меняются от 0 до 3, то параметрическая поверхность Безье задаётся следующим уравнением:

$$P(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 B_i^3(u) B_j^3(v) P_{ij}.$$

В приведённой выше формуле буквами В обозначены кубические многочлены Бернштейна (см. пример «Тесселяция плоской кривой»).

Нам также потребуется вычислить вектор нормали для каждой интерполированной точки. Для этого нам придётся вычислить векторное произведение частных производ-

ных $P(u, v)$:

$$\mathbf{n}(u, v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}.$$

Частные производные поверхности Безье сводятся к частным производным многочленов Бернштейна:

$$\frac{\partial P}{\partial u} = \sum_{i=0}^3 \sum_{j=0}^3 \frac{\partial B_i^3(u)}{\partial u} B_j^3(v) P_{ij},$$

$$\frac{\partial P}{\partial v} = \sum_{i=0}^3 \sum_{j=0}^3 B_i^3(u) \frac{\partial B_j^3(v)}{\partial v} P_{ij}.$$

Мы найдём производные в мозаичном вычисляющем шейдере и посчитаем векторное произведение, чтобы определить вектор нормали к поверхности в каждой вершине разбиения.

Подготовка

Создавайте свои шейдеры так, чтобы вершинный шейдер передавал положение вершины далее без изменений (вы можете использовать вершинный шейдер из примера «Тесселяция плоского четырёхугольника»). Создайте фрагментный шейдер, который реализует любой вариант освещённости и закраски по вашему выбору. На вход фрагментный шейдер должен принимать переменные `TENormal` и `TEPosition`, которые будут описывать нормаль и положение в системе отсчёта точки наблюдения.

Переменной типа `uniform TessLevel` нужно присвоить желаемое значение уровня разбиения. Все внутренние и внешние уровни примут это значение.

Как это сделать...

Чтобы создать программу шейдера, который будет создавать элементы поверхности по входным наборам из 16 контрольных точек, следуйте такому алгоритму:

Используйте вершинный шейдер из примера «Тесселяция плоского четырёхугольника».

Используйте следующий код в качестве мозаичного управляющего шейдера:

```
#version 400
layout( vertices=16 ) out;
uniform int TessLevel;
void main()
{
    // Передаём исходное положение вершин без изменения
    gl_out[gl_InvocationID].gl_Position =
    gl_in[gl_InvocationID].gl_Position;
    gl_TessLevelOuter[0] = float(TessLevel);
    gl_TessLevelOuter[1] = float(TessLevel);
    gl_TessLevelOuter[2] = float(TessLevel);
    gl_TessLevelOuter[3] = float(TessLevel);
    gl_TessLevelInner[0] = float(TessLevel);
    gl_TessLevelInner[1] = float(TessLevel);
}
```

Используйте следующий код в качестве мозаичного вычисляющего шейдера:

```
#version 400
layout( quads ) in;
out vec3 TENormal; // Нормаль к вершине в к-такх точки наблюдения
out vec4 TEPosition; // Положение в-ны в к-такх точки наблюдения
uniform mat4 MVP;
```

```

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
void basisFunctions(out float[4] b, out float[4] db, float t)
{
    float t1 = (1.0 - t);
    float t12 = t1 * t1;
    // Многочлены Бернштейна
    b[0] = t12 * t1;
    b[1] = 3.0 * t12 * t;
    b[2] = 3.0 * t1 * t * t;
    b[3] = t * t * t;
    // Производные
    db[0] = -3.0 * t1 * t1;
    db[1] = -6.0 * t * t1 + 3.0 * t12;
    db[2] = -3.0 * t * t + 6.0 * t * t1;
    db[3] = 3.0 * t * t;
}
void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    // 16 контрольных точек
    vec4 p00 = gl_in[0].gl_Position;
    vec4 p01 = gl_in[1].gl_Position;
    vec4 p02 = gl_in[2].gl_Position;
    vec4 p03 = gl_in[3].gl_Position;
    vec4 p10 = gl_in[4].gl_Position;
    vec4 p11 = gl_in[5].gl_Position;
    vec4 p12 = gl_in[6].gl_Position;
    vec4 p13 = gl_in[7].gl_Position;
    vec4 p20 = gl_in[8].gl_Position;
    vec4 p21 = gl_in[9].gl_Position;
    vec4 p22 = gl_in[10].gl_Position;
    vec4 p23 = gl_in[11].gl_Position;
    vec4 p30 = gl_in[12].gl_Position;
    vec4 p31 = gl_in[13].gl_Position;
    vec4 p32 = gl_in[14].gl_Position;
    vec4 p33 = gl_in[15].gl_Position;
    // Вычисление базисных функций
    float bu[4], bv[4]; // Базисные функции для u и v
    float dbu[4], dbv[4]; // Производные для u и v
    basisFunctions(bu, dbu, u);
    basisFunctions(bv, dbv, v);
    // Интерполяция поверхностью Безье
    TEPPosition =
        p00*bu[0]*bv[0] + p01*bu[0]*bv[1] + p02*bu[0]*bv[2] +
        p03*bu[0]*bv[3] +
        p10*bu[1]*bv[0] + p11*bu[1]*bv[1] + p12*bu[1]*bv[2] +
        p13*bu[1]*bv[3] +
        p20*bu[2]*bv[0] + p21*bu[2]*bv[1] + p22*bu[2]*bv[2] +
        p23*bu[2]*bv[3] +
        p30*bu[3]*bv[0] + p31*bu[3]*bv[1] + p32*bu[3]*bv[2] +
        p33*bu[3]*bv[3];
    // Частные производные
    vec4 du =
        p00*dbu[0]*bv[0] + p01*dbu[0]*bv[1] + p02*dbu[0]*bv[2] +
        p03*dbu[0]*bv[3] +
        p10*dbu[1]*bv[0] + p11*dbu[1]*bv[1] + p12*dbu[1]*bv[2] +
        p13*dbu[1]*bv[3] +
        p20*dbu[2]*bv[0] + p21*dbu[2]*bv[1] + p22*dbu[2]*bv[2] +
        p23*dbu[2]*bv[3] +
        p30*dbu[3]*bv[0] + p31*dbu[3]*bv[1] + p32*dbu[3]*bv[2] +
        p33*dbu[3]*bv[3];
    vec4 dv =
        p00*bu[0]*dbv[0] + p01*bu[0]*dbv[1] + p02*bu[0]*dbv[2] +
        p03*bu[0]*dbv[3] +
        p10*bu[1]*dbv[0] + p11*bu[1]*dbv[1] + p12*bu[1]*dbv[2] +
        p13*bu[1]*dbv[3] +

```

```

p20*bu[2]*dbv[0] + p21*bu[2]*dbv[1] + p22*bu[2]*dbv[2] +
p23*bu[2]*dbv[3] +
p30*bu[3]*dbv[0] + p31*bu[3]*dbv[1] + p32*bu[3]*dbv[2] +
p33*bu[3]*dbv[3];
// Вектор нормали есть векторное произведение частных производных
vec3 n = normalize(cross(du.xyz, dv.xyz));
// Переход к однородным координатам
gl_Position = MVP * TEPosition;
// Переход к координатам точки наблюдения
TEPosition = ModelViewMatrix * TEPosition;
TENormal = normalize(NormalMatrix * n);
}

```

Примените свой любимый вариант освещённости и закраски, используя выходные значения мозаичного вычисляющего шейдера.

Представьте контрольные точки поверхности Безье как примитив типа patch с 16-тью вершинами. Не забудьте задать число вершин примитива следующей командой:

`glPatchParameteri(GL_PATCH_VERTICES, 16);`

Как это работает...

Мозаичный управляющий шейдер начинается с задания числа вершин примитива следующей командой:

`layout (vertices=16) out;`

Затем он просто присваивает уровням разбиения значение переменной `TessLevel`. Положение вершин передаётся далее без изменений. Мозаичный вычисляющий шейдер начинается с команды `layout`, которая задаёт тип тесселяции. Так как мы собираемся проводить разбиение фрагмента, представляющего собой поверхность Безье, описываемую 16-тью точками, рациональнее всего использовать тесселяцию четырёхугольников.

Функция `basisFunctions` вычисляет многочлены Бернштейна и их производные для данного значения параметра t . Результаты присваиваются выходным переменным `bu` и `db`.

В основной функции мы начинаем с присвоения значений координат разбиения переменным `u` и `v`, а значения всех 16 вершин заново записываем в переменные с более короткими именами (чтобы сократить дальнейший код).

Затем мы вызываем функцию `basisFunctions`, чтобы вычислить многочлены Бернштейна и их производные по `u` и `v`, записывая результаты в переменные `bu`, `dbu`, `dv` и `dbv`.

Следующий шаг – вычисление сумм, как это было показано в вышеприведённых формулах для определения положения точек (`TEPosition`), и частных производных по `u` (`du`) и `v` (`dv`).

Мы вычисляем вектор нормали как векторное произведение `du` и `dv`.

На последнем этапе мы переводим положение точек (`TEPosition`) в однородные координаты и присваиваем результат переменной `gl_Position`. Перед передачей этих данных фрагментному шейдеру мы переводим их в систему отсчёта точки наблюдения.

Вектор нормали преобразуется в координаты точки наблюдения путём умножения на матрицу `NormalMatrix`, а результат нормируется и передаётся фрагментному шейдеру в переменной `TENormal`.

См. также

Тесселяция плоского четырёхугольника

Глава 4

Эффекты с помощью шейдеров

4.0.1 Эффект ряби (Ripple Effect)

Смещает пиксели в текстуре от середины к краям, создавая эффект ряби/волны (рис. 0.1).

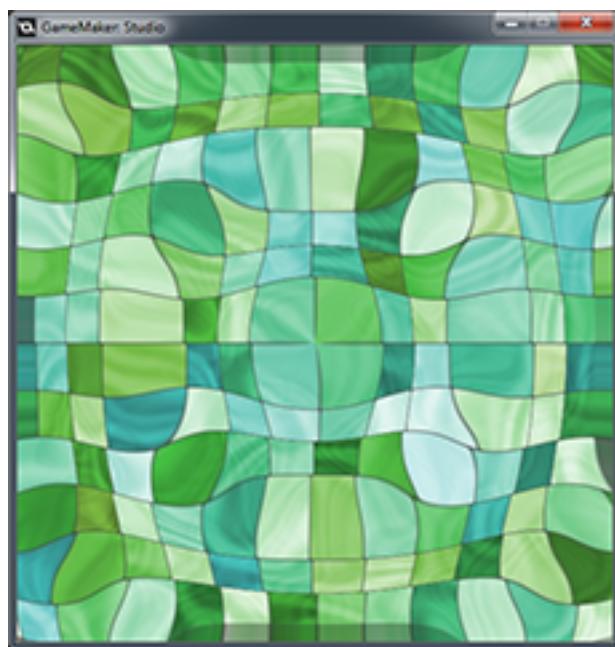


Рис. 0.1: Эффект ряби.

```
varying vec2 v_texcoord;
varying vec4 v_color;
uniform float time;

void main()
{
    vec2 tc = v_texcoord.xy;
    vec2 p = -1.0 + 2.0 * tc;
    float len = length(p);
    vec2 uv = tc + (p/len)*cos(len*12.0-time*4.0)*0.03;
    vec3 col = texture2D(gm_BaseTexture,uv).xyz;
    gl_FragColor = vec4(col,1.0);
}
```

4.0.2 Эффект волны (Wave Effect)

Смещает пиксели в текстуре подобно волне, создавая эффект воды/лавы (рис. 0.2).

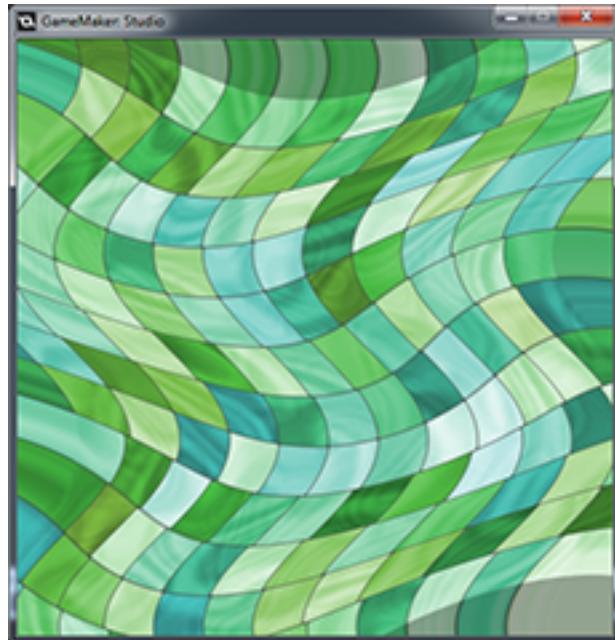


Рис. 0.2: Эффект волны.

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform float time;
void main()
{
    vec2 texCoord = v_texcoord;
    texCoord.x = texCoord.x+cos(texCoord.y*10.0+time*0.05)*0.1;
    texCoord.y = texCoord.y+sin(texCoord.x*5.0+time*0.05)*0.1;
    gl_FragColor = texture2D(gm_BaseTexture, texCoord);
}
}
    
```

4.0.3 Эффект тиснения (Emboss Effect using sprite and alpha)

Эффект тиснения, с помощью спрайта и альфа-маски (рис. 0.3).

```

varying vec2 v_vTexcoord;
varying vec4 v_vColour;
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

void main()
{
    vec2 onePixel = vec2(1.0 / 512.0, 1.0 / 512.0);
    vec2 texCoord = v_vTexcoord;  vec4 colour;
    colour.rgb = vec3(0.5);
    colour -= texture2D(gm_BaseTexture, texCoord - onePixel) * 5.0;
    colour += texture2D(gm_BaseTexture, texCoord + onePixel) * 5.0;
    colour.rgb = vec3((colour.r + colour.g + colour.b) / 3.0);
    gl_FragColor = vec4(colour.rgb, texture2D(gm_BaseTexture,v_vTexcoord).a);
}
    
```



Рис. 0.3: Эффект тиснения.

4.0.4 Эффект ударной волны (Shockwave Effect)

Создаёт ударную волну, движущуюся противоположно от положения курсора (рис. 0.4).

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform vec2 center;
uniform float time;
uniform vec3 shockParams;

void main()
{
    vec2 uv = v_texcoord.xy;
    vec2 texCoord = uv;
    float dist = distance(uv, center);

    if ( (dist <= (time + shockParams.z)) && (dist >= (time - shockParams.z)) )
    {
        float dif = (dist - time);
        float powDiff = 1.0 - pow(abs(dif*shockParams.x), shockParams.y);
        float diffTime = dif * powDiff;      vec2 diffUV = normalize(uv - center);
        texCoord = uv + (diffUV * diffTime);
    }

    gl_FragColor = texture2D(gm_BaseTexture, texCoord);
}

```

4.0.5 Эффект телевизора (TV Effect)

Строки, мерцания, помехи, подсветка. Строки движутся вверх, как в эффекте голограммы (рис. 0.5).

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform float time;
vec2 resolution = vec2(1.0,1.0);

```

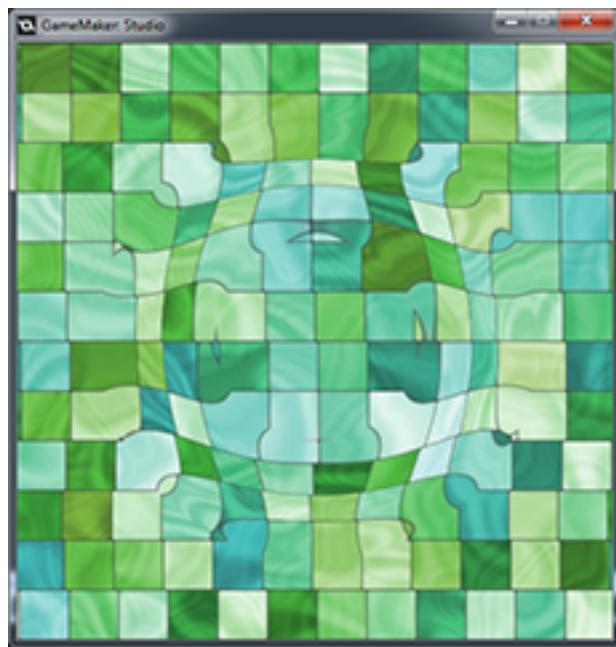


Рис. 0.4: Эффект ударной волны.

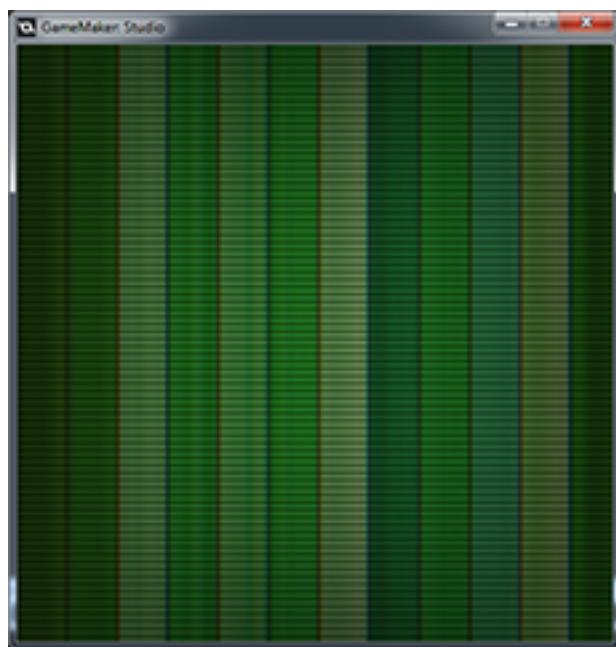


Рис. 0.5: Эффект телевизора.

```

float rand(vec2 co) {
    return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453);
}

void main()
{
    vec2 q1 = v_texcoord.xy / resolution.xy;
    vec2 uv = 0.5 + (q1-0.5)*(0.98 + 0.006); /*sin (0.9));
    vec3 oricol = texture2D(gm_BaseTexture ,vec2(q1.x,1.0-q1.y)).xyz;
    vec3 col;
    col.r = texture2D(gm_BaseTexture ,vec2(uv.x+0.003,uv.y)).x;
    col.g = texture2D(gm_BaseTexture ,vec2(uv.x+0.000,uv.y)).y;
    col.b = texture2D(gm_BaseTexture ,vec2(uv.x-0.003,uv.y)).z;
    col = clamp(col*0.5+0.5*col*col*1.2,0.0,1.0);
    col *= 0.6 + 0.4*16.0*uv.x*uv.y*(1.0-uv.x)*(1.0-uv.y);
    col *= vec3(0.9,1.0,0.7);
    col *= 0.8+0.2*sin(10.0*time+uv.y*512.0);
    col *= 1.0-0.07*rand(vec2(time, tan(time)));
    float comp = smoothstep( 0.2, 0.7, sin(time) );
    gl_FragColor = vec4(col,1.0);
}

```

4.0.6 Эффект яркости/выцветания (Brightness/Bloom Effect).

См. рис. 0.6.

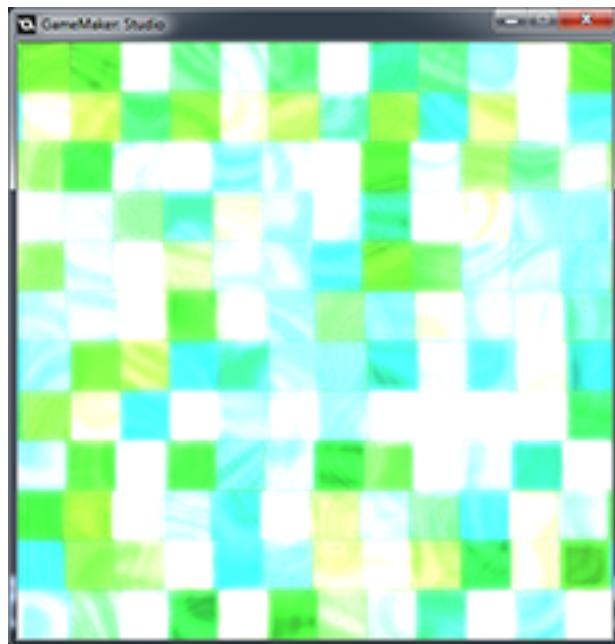


Рис. 0.6: Эффект яркости/выцветания.

```

varying vec2 v_texcoord;
varying vec4 v_color;

void main()
{
    vec4 sum = vec4(0.0);
    vec2 q1 = v_texcoord;
    vec4 oricol = texture2D(gm_BaseTexture ,vec2(q1.x,q1.y));
    vec3 col;

    for(int i=-4;i<4;i++) {

```

```

        for (int j=-3;j<3;j++) {
            sum += texture2D(gm_BaseTexture ,vec2(j,i)*0.004 +
                vec2(q1.x,q1.y)) * 0.25;
        }
    }

    if (oricol.r < 0.4) {
        gl_FragColor = sum*sum*0.012 + oricol;
    } else {
        if (oricol.r < 0.6) {
            gl_FragColor = sum*sum*0.009 + oricol;
        } else {
            gl_FragColor = sum*sum*0.0075 + oricol;
        }
    }
}

```

4.0.7 Рельефное текстурирование «Normal Mapping»

Создаёт рельефные светотени от источника света в координатах мыши (рис. 0.7).

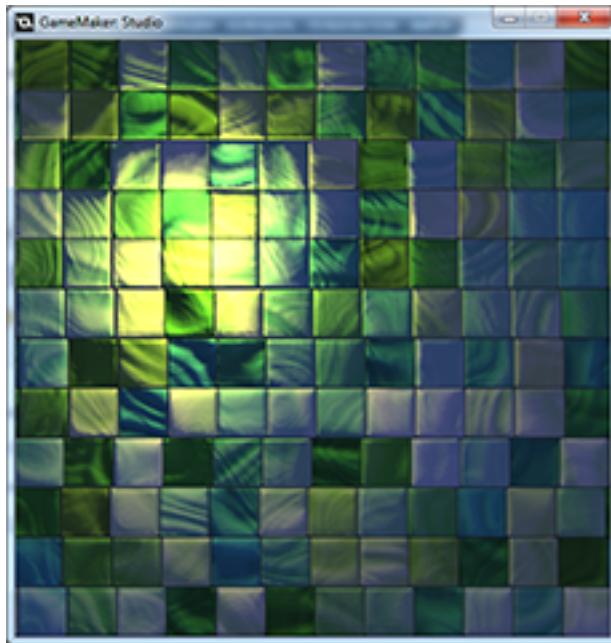


Рис. 0.7: Эффект освещения с рельефным текстурированием.

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform sampler2D s_multitex; //values used for shading algorithm...
uniform vec2 Resolution; //resolution of screen
uniform vec3 LightPos; //light position, normalized
uniform vec4 LightColor; //light RGBA — alpha is intensity
uniform vec4 AmbientColor; //ambient RGBA — alpha is intensity
uniform vec3 Falloff; //attenuation coefficients
main(){ //RGBA of our diffuse color
    vec4 DiffuseColor = texture2D(gm_BaseTexture , v_texcoord); //RGB карты нормалей
    vec3 NormalMap = texture2D(s_multitex , v_texcoord).rgb; //delta-положение света

    //Исправляем соотношение сторон
    vec3 LightDir = vec3(LightPos.xy - (v_texcoord.xy / Resolution.xy), LightPos.z);

    //Determine distance (used for attenuation) BEFORE we normalize our LightDir
}

```

```

LightDir.x *= Resolution.x / Resolution.y;
float D = length(LightDir);           //normalize our vectors
vec3 N = normalize(NormalMap * 2.0 - 1.0);
vec3 L = normalize(LightDir);         //Pre-multiply light color with intensity

//Then perform "N dot L" to determine our diffuse term

//pre-multiply ambient color with intensity
vec3 Diffuse = (LightColor.rgb * LightColor.a) * max(dot(N, L), 0.0);
vec3 Ambient = AmbientColor.rgb * AmbientColor.a;    //calculate attenuation

//the calculation which brings it all together
float Attenuation = 1.0 / (Falloff.x + (Falloff.y*D) + (Falloff.z*D*D));
vec3 Intensity = Ambient + Diffuse * Attenuation;
vec3 FinalColor = DiffuseColor.rgb * Intensity;
gl_FragColor = v_color * vec4(FinalColor, DiffuseColor.a);}
```

4.0.8 Эффект анимации травы (Grass Animation Effect)

Анимирует изображения, передвигая пиксели, двигает траву подобно волне назад и вперёд (рис. 0.8).

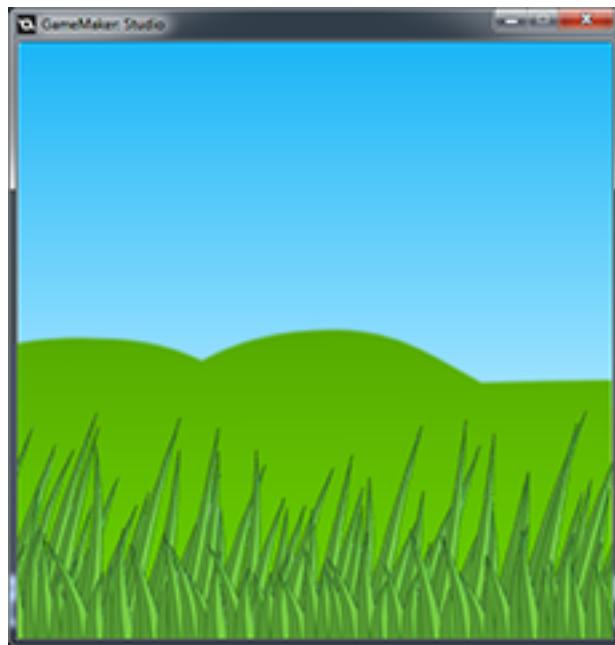


Рис. 0.8: Эффект анимации травы.

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform float time;
uniform float spd;
uniform float bendfactor;

void main()
{
    float height = 1.0 - v_texcoord.y;
    float offset = pow(abs(height), 1.0);
    offset *= (sin(time * spd) * bendfactor);
    vec4 normalColor = texture2D(gm_BaseTexture,
                                  fract(vec2(v_texcoord.x + offset, v_texcoord.y))).rgba;
    gl_FragColor = normalColor;
```

```
| }
```

4.0.9 Эффект голограммы (Hologram Effect)

Создаёт эффект голограммы передвигая линии на изображении вверх и подсвечивая её область (рис. 0.9).

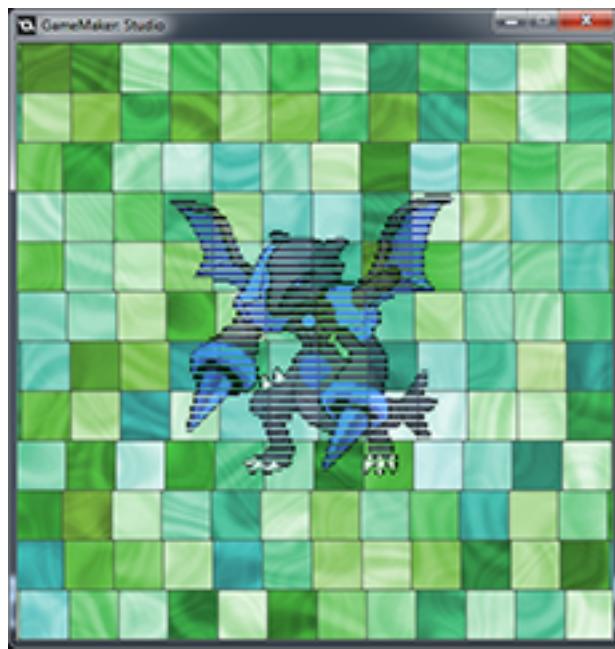


Рис. 0.9: Эффект голограммы.

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform float Stripes;
uniform vec4 Filter;
uniform float Phase;

vec4 HologramColor( vec4 colour, float stripes, vec4 filter, float phase )
{
    colour.a *= abs( sin( radians( v_texcoord.y * 180.0 * stripes + phase * 180.0 ) ) );
    return( colour * filter );
}

vec4 FixedColor( sampler2D texture, vec4 col )
{
    vec4 colour = texture2D( texture, v_texcoord.xy );return( colour * col );
}

void main()
{
    gl_FragColor = HologramColor( FixedColor( gm_BaseTexture, v_color ),
                                Stripes, Filter, Phase );
}

```

4.0.10 Эффект воды для TDS.

Совмещение нескольких, ранее представленных, средств (рис. 0.10).

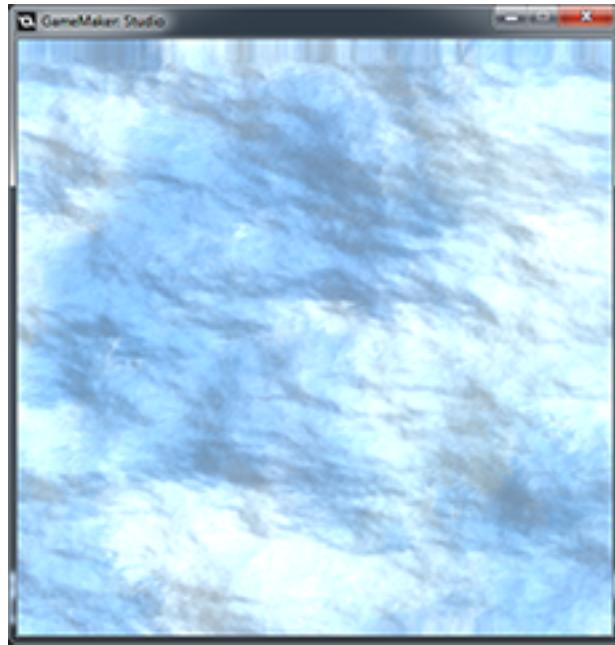


Рис. 0.10: Эффект воды.

4.0.11 Эффект Гауссова размытия (Gaussian Blur Effect)

См. рис. 0.11.

```

varying vec2 v_vTexCoord;
varying vec4 v_vColour;

// The sigma value for the gaussian function: higher value means more blur
uniform float sigma;
// A good value for 9x9 is around 3 to 5
// A good value for 7x7 is around 2.5 to 4
// A good value for 5x5 is around 2 to 3.5
// ... play around with this based on what you need :)
uniform float blurSize; // This should usually be equal to

// 1.0f / texture_pixel_width for a horizontal blur, and
// 1.0f / texture_pixel_height for a vertical blur.

float pii = 3.14159265;
float numBlurPixelsPerSide = 3.0;
vec2 blurMultiplyVec = vec2(1.0, 0.0);

void main()
{
    // Incremental Gaussian Coefficient Calculation (See GPU Gems 3 pp. 877 – 889)
    vec3 incrementalGaussian;
    incrementalGaussian.x = 1.0 / (sqrt(2.0 * pii) * sigma);
    incrementalGaussian.y = exp(-0.5 / (sigma * sigma));
    incrementalGaussian.z = incrementalGaussian.y * incrementalGaussian.y;
    vec4 avgValue = vec4(0.0, 0.0, 0.0, 0.0); float coefficientSum = 0.0;
    // Take the central sample first...
    avgValue += texture2D(gm_BaseTexture, v_vTexCoord.xy) * incrementalGaussian.x;
    coefficientSum += incrementalGaussian.x;
    incrementalGaussian.xy *= incrementalGaussian.yz;

    // Go through the remaining 8 vertical samples (4 on each side of the center)
    for (float i = 1.0; i <= numBlurPixelsPerSide; i++)
    {
        avgValue += texture2D(gm_BaseTexture, v_vTexCoord.xy -

```

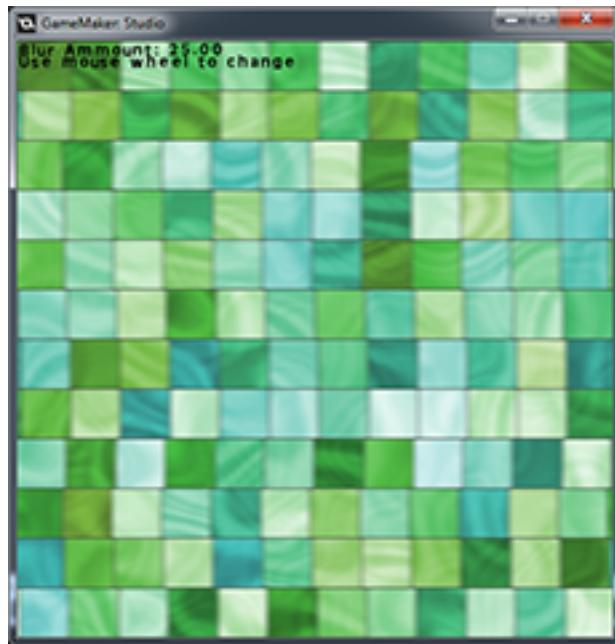


Рис. 0.11: Эффект Гауссова размытия.

```

        i * blurSize * blurMultiplyVec) * incrementalGaussian.x;
avgValue += texture2D(gm_BaseTexture, v_vTexCoord.xy +
        i * blurSize * blurMultiplyVec) * incrementalGaussian.x;
coefficientSum += 2.0 * incrementalGaussian.x;
incrementalGaussian.xy *= incrementalGaussian.yz;
}

gl_FragColor = avgValue / coefficientSum;
}

```

4.0.12 Эффект старого фильма / Сепии (Old Film Effect)

Стиль старой кинокамеры, может использовать текстурное закрашивание, изменение оттенка, шумы и царапины (рис. 0.12).

```

uniform float g_Time;
uniform vec4 m_FilterColor;
uniform float m_ColorDensity;
uniform float m_RandomValue;
uniform float m_NoiseDensity;
uniform float m_ScratchDensity;
varying vec2 texCoord;
varying vec4 v_color;

// Computes the overlay between the source and destination colours.

vec3 overlay(vec3 src, vec3 dst)
{
float aa, bb, cc;
if (dst.x <= 0.5) {aa = (2.0 * src.x * dst.x);
}

```

4.0.13 Радиальное размытие (Radial Blur)

Радиальное размытие (рис. 0.13).



Рис. 0.12: Эффект старого фильма.

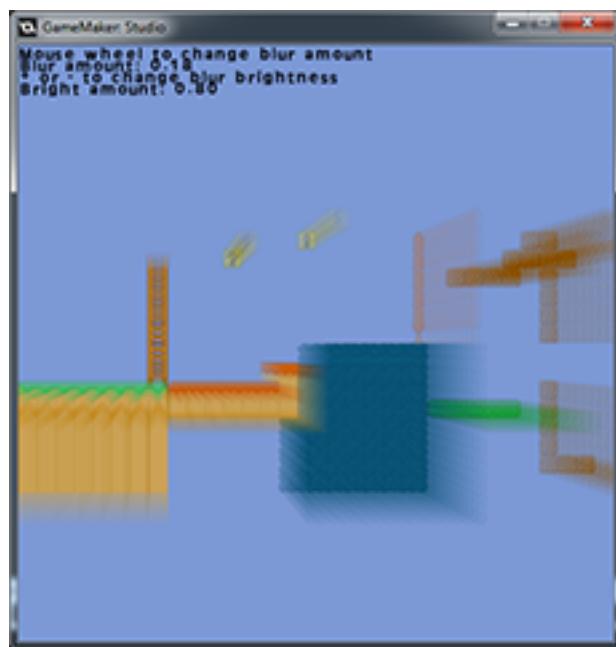


Рис. 0.13: Радиальное размытие.

```

uniform float g_Time;
uniform vec4 m_FilterColor;
uniform float m_ColorDensity;
uniform float m_RandomValue;
uniform float m_NoiseDensity;
uniform float m_ScratchDensity;
varying vec2 texCoord;
varying vec4 v_color;

// Рассчитываем наложение цветов источника и приёмника
vec3 overlay(vec3 src, vec3 dst) {
    float aa, bb, cc;

    if (dst.x <= 0.5) {aa = (2.0 * src.x * dst.x);}
    else {aa = (1.0 - 2.0 * (1.0 - dst.x)) * (1.0 - src.x);}

    if (dst.y <= 0.5) {bb = (2.0 * src.y * dst.y);}
    else {bb = (1.0 - 2.0 * (1.0 - dst.y)) * (1.0 - src.y);}

    if (dst.z <= 0.5) {cc = (2.0 * src.z * dst.z);}
    else {cc = (1.0 - 2.0 * (1.0 - dst.z)) * (1.0 - src.z);}

    return vec3(aa, bb, cc);
}

// 2D Noise by Ian McEwan, Ashima Arts.
vec3 mod289(vec3 xx) { return xx - ffloor(xx * (1.0 / 289.0)) * 289.0; }
vec2 mod289(vec2 xx) { return xx - ffloor(xx * (1.0 / 289.0)) * 289.0; }
vec3 permute(vec3 xx) { return mod289(((xx*34.0)+1.0)*xx); }

float snoise (vec2 v) {
    const vec4 C = vec4(0.211324865405187, // (3.0-sqrt(3.0))/6.0
                        0.366025403784439, // 0.5*(sqrt(3.0)-1.0)
                        -0.577350269189626, // -1.0 + 2.0 * C.x
                        0.024390243902439); // 1.0 / 41.0

    // Первый угол
    vec2 i = ffloor(v + dot(v, C.yy));
    vec2 x0 = v - i + dot(i, C.xx);

    // Прочие углы
    vec2 i1;
    i1 = (x0.x > x0.y) ? vec2(1.0, 0.0) : vec2(0.0, 1.0);
    vec4 x12 = x0.xyxy + C.xxzz;
    x12.xy -= i1;

    // Перестановки
    i = mod289(i); // Avoid truncation effects in permutation
    vec3 pp = permute(permute(i.y + vec3(0.0, i1.y, 1.0)) + i.x +
                      vec3(0.0, i1.x, 1.0));
    vec3 m = max(0.5 - vec3(dot(x0, x0), dot(x12.xy, x12.xy), dot(x12.zw, x12.zw)), 0.0);
    m = m*m;
    m = m*m;

    // Gradients: 41 points uniformly over a line, mapped onto a diamond.
    // The ring size 17*17 = 289 is close to a multiple of 41 (41*7 = 287)

    vec3 xx = 2.0 * fract(pp * C.www) - 1.0;
    vec3 h = abs(xx) - 0.5;
    vec3 ox = floor(xx + 0.5);
    vec3 a0 = xx - ox;

    // Normalise gradients implicitly by scaling m
    // Approximation of: m *= inversesqrt( a0*a0 + h*h );
    m *= 1.79284291400159 - 0.85373472095314 * (a0*a0 + h*h);

    // Вычисляем конечный шум в P
}

```

```

    vec3 gg;
    gg.x = a0.x * x0.x + h.x * x0.y;
    gg.yz = a0.yz * x12.xz + h.yz * x12.yw;
    return 130.0 * dot(m, gg);
}

void main() {
    // Convert to grayscale
    vec3 colour = texture2D(gm_BaseTexture, texCoord).rgb;
    float gray = (colour.r + colour.g + colour.b) / 3.0;
    vec3 grayscale = vec3(gray);

    // Apply overlay
    vec3 finalColour = overlay(m_FilterColor.rgb, grayscale);

    // Lerp final colour
    float colorFactor = clamp(m_ColorDensity, 0.0, 1.0);
    finalColour = grayscale + colorFactor * (finalColour - grayscale);

    // Add noise
    float noiseFactor = clamp(m_NoiseDensity, 0.0, 1.0);
    float noise = snoise(texCoord * vec2(1024.0 + m_RandomValue * 512.0,
                                          1024.0 + m_RandomValue * 512.0)) * 0.5;
    finalColour += noise * noiseFactor;

    // Apply scratches
    float scratchFactor = clamp(m_ScratchDensity, 0.0, 1.0);
    if (m_RandomValue < scratchFactor)
    {
        // Pick a random spot to show scratches
        float dist = 1.0 / scratchFactor;
        float d = distance(texCoord, vec2(m_RandomValue * dist, m_RandomValue * dist));
        if (d < 0.4) {
            // Generate the scratch
            float xPeriod = 8.0;
            float yPeriod = 1.0;
            float pii = 3.141592;
            float phase = g_Time;
            float turbulence = snoise(texCoord * 2.5);
            float vScratch = 0.5 + (sin(((texCoord.x * xPeriod +
                                         texCoord.y * yPeriod + turbulence)) * pii + phase) * 0.5);
            vScratch = clamp((vScratch * 10000.0) + 0.35, 0.0, 1.0);
            finalColour.xyz *= vScratch;
        }
    }

    // Apply colour
    //use this if you want color
    //gl_FragColor.xyz = finalColour * texture2D(gm_BaseTexture, texCoord).rgb;
    gl_FragColor.xyz = finalColour; //User this if you want grey scale
    gl_FragColor.w = 1.0;
}

```

4.0.14 Эффект воды для платформера

Эффект воды для платформера (рис. 0.14).

```

uniform vec4 u_vWaterParams;
uniform vec4 u_vWaterParams2;
uniform vec4 u_vWaterParams3;
uniform vec4 u_vSineParams;
uniform vec4 u_vSineParams2;
uniform vec4 u_vRefractColour;
uniform vec4 u_vReflectColour;
varying vec2 v_texcoord;
varying vec4 v_color;

```

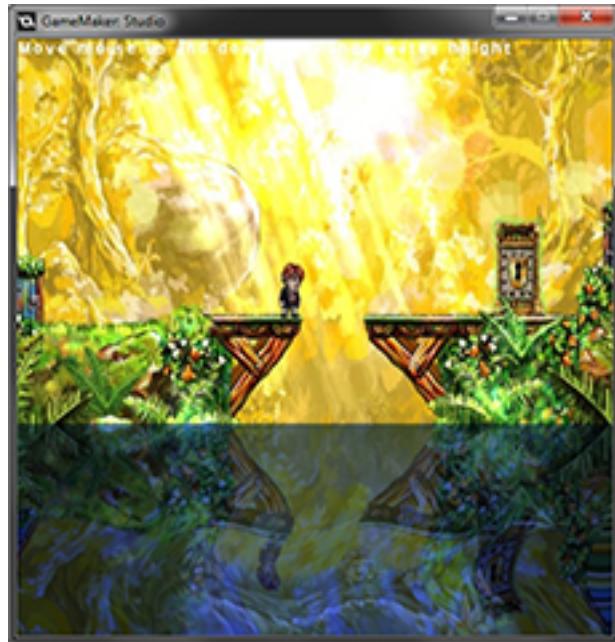


Рис. 0.14: Эффект воды для платформера.

```

uniform sampler2D s_TexSampler;

void main() {
    vec4 vertColour = v_color;
    vec2 texcoord = v_texcoord;
    vec4 combinedColour;
    if (texcoord.y > u_vWaterParams.x) {
        float offset = texcoord.y - u_vWaterParams.x;
        float mag = offset * u_vWaterParams.y;

        //float sinescale = u_vWaterParams3.x - (offset * u_vWaterParams3.y);
        //sinescale = clamp(sinescale, 1.0, u_vWaterParams3.x);
        float sinescale = 1.0;

        vec2 shift1, shift2;

        shift1.x = sin((offset * u_vSineParams.x * sinescale) + u_vSineParams.y);
        shift1.y = sin((texcoord.x * u_vSineParams.x) + u_vSineParams.z);

        shift2.x = sin((offset * u_vSineParams2.x * sinescale) + u_vSineParams2.y);
        shift2.y = sin((texcoord.x * u_vSineParams2.x) + u_vSineParams2.z);

        //vertColour += (shift1.x * 0.1) + (shift1.y * 0.1) + (shift2.x * 0.1) +
        //               (shift2.y * 0.1);

        texcoord.x += shift1.x * u_vWaterParams.z * mag;
        texcoord.y += shift1.y * u_vWaterParams.w * mag;

        texcoord.x += shift2.x * u_vWaterParams2.z * mag;
        texcoord.y += shift2.y * u_vWaterParams2.w * mag;
        //
        vec4 refractTexColour = texture2D( s_TexSampler, texcoord );
        vec4 reflectTexColour = texture2D( s_TexSampler,
            vec2(texcoord.x, u_vWaterParams.x - (texcoord.y - u_vWaterParams.x)));
        reflectCol = reflectTexColour * u_vReflectColour;
        reflectCol *= 1.0 - (mag * u_vWaterParams2.x);
        reflectCol = clamp(reflectCol, 0.0, 1.0);
        combinedColour = vertColour *

```

```

        ((refractTexColour * u_vRefractColour) + reflectCol);
    } else {
        vec4 texelColour = texture2D( s_TexSampler, v_texcoord );
        combinedColour = vertColour * texelColour;
    }
    gl_FragColor = combinedColour;
}

```

4.0.15 Эффект лучей проходящих сквозь препятствия

Эффект лучей проходящих сквозь препятствия / «божественного света» (God Rays Shader).

Динамический световой движок с эффектом "God Rays" (рис. 0.15).



Рис. 0.15: Эффект лучей проходящих сквозь препятствия.

```

varying vec2 v_vTexCoord;
varying vec4 v_vColour;
uniform vec2 lightPositionOnScreen;
uniform vec2 surfaceSize;
uniform float Density;
uniform float Weight;
uniform float Decay;
uniform float Exposure;
const int NUM_SAMPLES = 100 ;
void main()
{
    vec2 deltaTexCoord = vec2(v_vTexCoord -
        vec2(lightPositionOnScreen.x / surfaceSize.x,
              lightPositionOnScreen.y / surfaceSize.y));
    vec2 textCoo = v_vTexCoord;
    deltaTexCoord *= 1.0 / float(NUM_SAMPLES) * Density;
    float illuminationDecay = 1.0;

    for(int i=0; i < NUM_SAMPLES ; i++)
    {
        textCoo -= deltaTexCoord;
        vec4 sample = texture2D(gm_BaseTexture, textCoo );

```

```

        sample *= illuminationDecay * Weight;
        gl_FragColor += sample;
        illuminationDecay *= Decay;
    }

    gl_FragColor *= Exposure;
}

```

4.0.16 Эффект стекла / витража (Stained Glass Shader)

Эффект стекла (рис. 0.16).

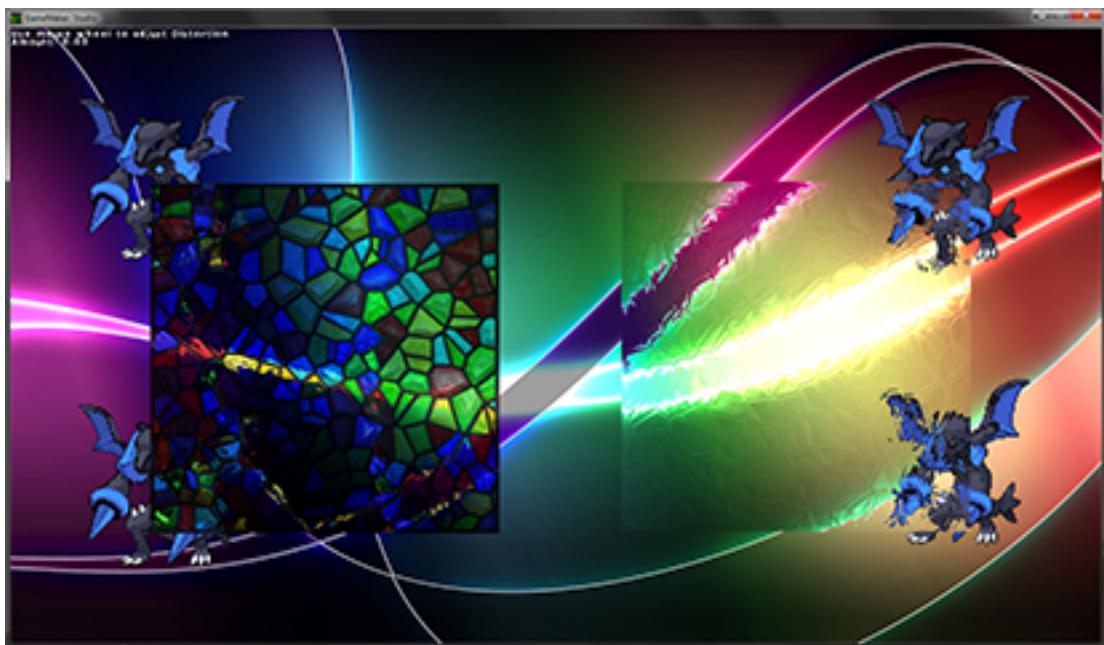


Рис. 0.16: Эффект стекла / витража.

```

varying vec2 v_texcoord;
varying vec4 v_color;

uniform sampler2D samp_Normal_Map;
uniform sampler2D samp_Diffuse_Tex;

uniform vec2 Room_Res;           //resolution of screen / room
uniform vec2 Texture_Res;        //resolution of texture / sprite / normal map
uniform vec3 Texture_Pos;        //Texture position (world pos)
uniform float Refraction_Index; //approx refraction strength see

void main()
{
    vec3 downRay = vec3(0.5, 0.5, 1.0);
    vec2 imagePos = v_texcoord - (Texture_Pos.xy / Room_Res.xy);
    vec2 aspectRatio = vec2(Room_Res.x / Texture_Res.x, Room_Res.y / Texture_Res.y);
    vec3 NormalMap = texture2D(samp_Normal_Map, vec2(imagePos.x*aspectRatio.x,
                                                       imagePos.y*aspectRatio.y)).rgb;
    vec3 normal = normalize(NormalMap);
    vec3 refractVec = downRay + Refraction_Index * normal;
    float vecScale = Texture_Pos.z / -refractVec.z;
    vec3 offset = (vecScale * refractVec);
    vec4 diffuse_tex = texture2D(samp_Diffuse_Tex, vec2(imagePos.x*aspectRatio.x,
                                                       imagePos.y*aspectRatio.y));
    vec4 refractPixel = texture2D(gm_BaseTexture, v_texcoord + offset.xy);

```

```

    gl_FragColor = diffuse_tex * refractPixel;
}

```

4.0.17 Эффект светлячков/блёстков/сияний (Happy Fireflies)

См. рис. 0.17.

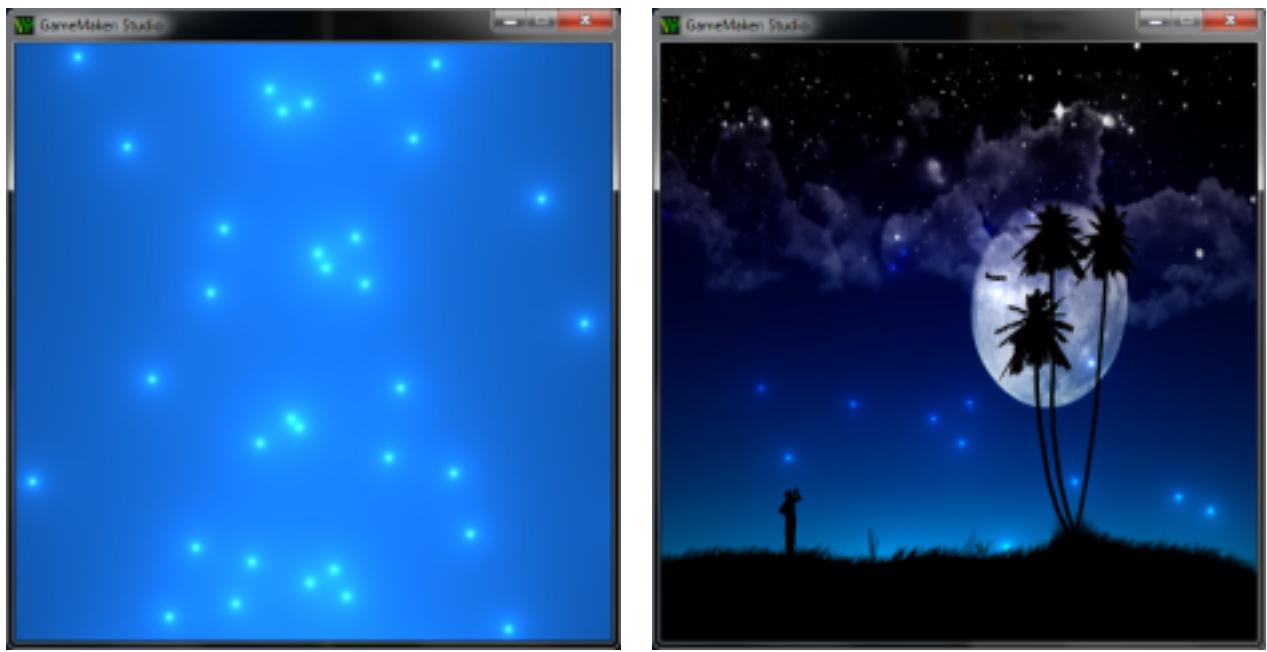


Рис. 0.17: Эффект светлячков/блёстков/сияний.

```

varying vec2 v_texcoord;
varying vec2 v_texcoord;
varying vec4 v_color;
uniform float time;
const int num = 100;
void main()
{
    float sum = 0.0;
    float size = 0.005;
    for (int i = 0; i < num; ++i) {
        vec2 pos = vec2(1.0,1.0)/2.0;
        float t = (float(i) + time) / 5.0;
        float c = float(i);
        pos.x += tan(4.0 * t + c) * 0.2;
        pos.y += sin(t) * 0.8;
        sum += size / length(v_texcoord.xy - pos);
    }
    //If you want to use a texture use these 3 line
    vec3 texture = texture2D(gm_BaseTexture,v_texcoord).rgb;
    vec3 final = texture2D(gm_BaseTexture,v_texcoord).rgb *
        vec3(sum * texture.r, sum * texture.g, sum * texture.b);
    gl_FragColor = vec4(final,v_color.a);

    //Don't want a texture, just use this line
    //gl_FragColor = vec4(sum * 0.1, sum * 0.5, sum, 1.0);
}

```

4.0.18 Эффект огненного вакуума/фаярболла (Fire Vacuum)

Смещает пиксели от краёв к середине, создавая эффект "всасывания" огня (рис. 0.18).

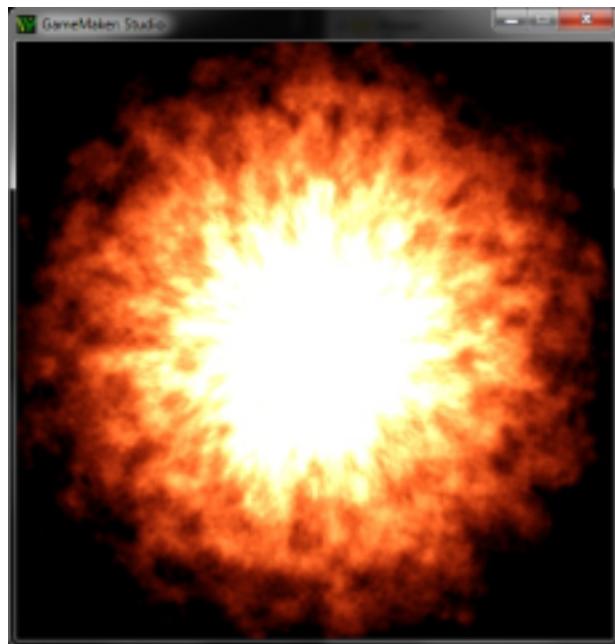


Рис. 0.18: Эффект огненного вакуума.

4.0.19 Эффект подводного освещения

Эффект подводного освещения (с анимацией) (2D Underwater God Rays)

Шейдер подводного эффекта God Rays с небольшой анимацией (рис. 0.19).

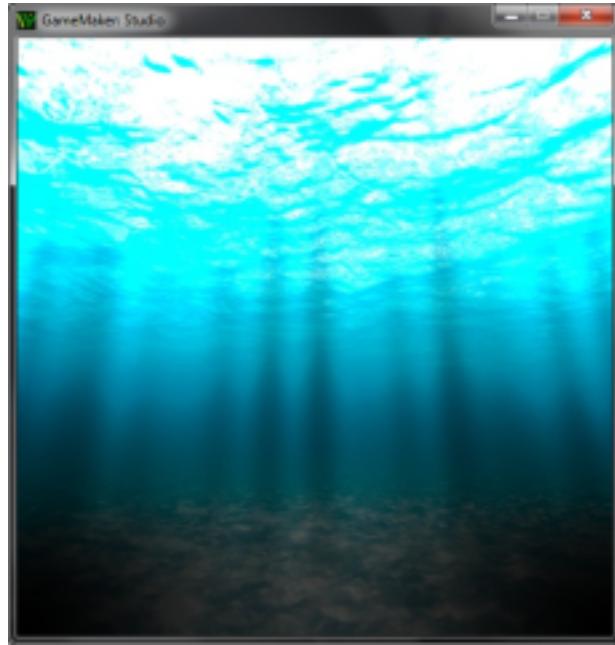


Рис. 0.19: Эффект подводного освещения.

```

varying vec2 v_texcoord;
varying vec4 v_color;
uniform float time;
float CausticPatternFn(vec2 pos)
{
    return (sin(pos.x*40.0+time)
        +pow(sin(-pos.x*130.0+time),1.0)
        +pow(sin(pos.x*30.0+time),2.0)
        +pow(sin(pos.x*50.0+time),2.0)
        +pow(sin(pos.x*80.0+time),2.0)
        +pow(sin(pos.x*90.0+time),2.0)
        +pow(sin(pos.x*12.0+time),2.0)
        +pow(sin(pos.x*6.0+time),2.0)
        +pow(sin(-pos.x*13.0+time),5.0))/2.0;
}

vec2 CausticDistortDomainFn(vec2 pos)
{
    pos.x+=(pos.y*0.20+0.5);
    pos.x+=1.0+sin(time/1.0)/10.0;
    return pos;
}

void main()
{
    vec2 pos = -v_texcoord.xy;
    pos += 0.5;
    vec2 CausticDistortedDomain = CausticDistortDomainFn(pos);
    float CausticShape = clamp(7.0-length(CausticDistortedDomain.x*20.0),0.0,1.0);
    float CausticPattern = CausticPatternFn(CausticDistortedDomain);
    float Caustic;
    Caustic += CausticShape*CausticPattern;
    Caustic *= (pos.y+0.5)/4.0;
}

```

```

//use these 2 line with a texture
float f = length(pos+vec2(-0.5,0.5))*length(pos+vec2(0.5,0.5))*(1.0+Caustic)/0.5;
//lower the last device to change the alpha
gl_FragColor = texture2D(gm_BaseTexture,v_texcoord) * f; //use with texture

//use these 2 line with a WITHOUT texture
//float f = length(pos+vec2(-0.5,0.5))*length(pos+vec2(0.5,0.5))*(1.0+Caustic)/1.0;
//set last devide (alpah) back to 1.0
//gl_FragColor = vec4(.1,.5,.6,1) * f; //use without a texture

}

```

4.0.20 Мультитекстурирование (Multi Texturing)

Позволяет нанести несколько текстур (рис. 0.20).

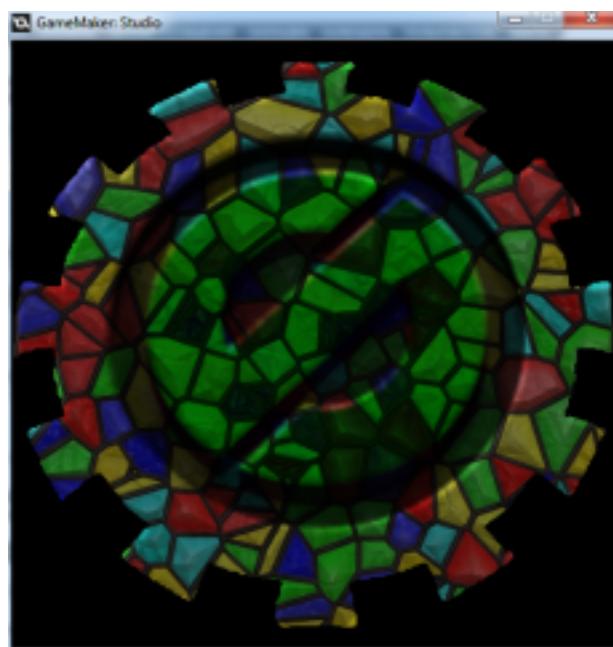


Рис. 0.20: Мультитекстурирование.

```

varying vec2 v_texcoord;
uniform sampler2D s_multitex;
void main()
{
    vec4 color1 = texture2D(gm_BaseTexture,v_texcoord);
    vec4 color2 = texture2D(s_multitex,v_texcoord);
    vec3 blendcolor = color1.rgb * color2.rgb;
    gl_FragColor = vec4(blendcolor, color1.a);
}

```

4.0.21 Эффект черно-белого изображения (Grey Scale Effect)

Сделано с учётом особенностей восприятия различных цветов - разные цвета с разной яркостью (рис. 0.21).

```

// Swap color values around

```



Рис. 0.21: Эффект черно-белого изображения.

```
//  
varying vec2 v_vTexCoord;  
varying vec4 v_vColour;  
  
void main()  
{  
    vec4 colour = texture2D( gm_BaseTexture, v_vTexCoord );  
    gl_FragColor = vec4( vec3(dot(colour.rgb, vec3(0.299, 0.587, 0.114))), colour.a );  
}
```

4.0.22 Эффект мозаики / пикселизации (Mosaic Effect)

Эффект пикселизации с настраиваемым размером ячейки (рис. 0.22).



Рис. 0.22: Эффект мозаики / пикселизации.

```

varying vec2 v_vTexcoord;

uniform float pixel_amount;
uniform vec2 resolution;

void main()
{
    vec2 res = vec2(1.0, resolution.x/resolution.y);
    vec2 size = vec2(res.x/pixel_amount, res.y/pixel_amount);
    vec2 Pbase = v_vTexcoord - mod(v_vTexcoord, size);
    gl_FragColor = texture2D( gm_BaseTexture, Pbase );
}

```

4.0.23 Эффект линзы / «рыбий глаз» (Magnify Effect).

Эффект линзы / «рыбий глаз» (рис. 0.23).



Рис. 0.23: Эффект линзы / «рыбий глаз».

```

varying vec2 v_texcoord;
varying vec4 v_color;

uniform vec2 center;
uniform vec2 resolution;

float circleRadius = float(0.25);
float minZoom = 0.4;
float maxZoom = 0.6;

void main()
{
    vec2 uv = v_texcoord;
    uv.x *= (resolution.x/resolution.y);

    vec2 aspect_center = vec2(0.0, 0.0);
    aspect_center.x = (center.x / resolution.x) * (resolution.x/resolution.y);
    aspect_center.y = center.y / resolution.y;
}

```

```
float maxX = aspect_center.x + circleRadius;
float minX = aspect_center.x - circleRadius;
float maxY = aspect_center.y + circleRadius;
float minY = aspect_center.y - circleRadius;

if( uv.x > minX && uv.x < maxX && uv.y > minY && uv.y < maxY)
{
    float relX = uv.x - aspect_center.x;
    float relY = uv.y - aspect_center.y;
    float ang = atan(relY, relX);
    float dist = sqrt(relX*relX + relY*relY);

    if( dist <= circleRadius )
    {
        float newRad = dist * ( (maxZoom * dist/circleRadius) + minZoom );
        float newX = aspect_center.x + cos( ang ) * newRad;
        newX *= (resolution.y/resolution.x);
        float newY = aspect_center.y + sin( ang ) * newRad;
        gl_FragColor = texture2D(gm_BaseTexture, vec2(newX, newY) );
    }
    else
    {
        gl_FragColor = texture2D(gm_BaseTexture, v_texcoord);
    }
}
else
{
    gl_FragColor = gl_FragColor = texture2D(gm_BaseTexture, v_texcoord);
}
```

Глава 5

Рельефное текстурирование

5.0.1 Рельефное текстурирование

Наблюдатель видит детали поверхности, когда рассматривает её в различных направлениях. Освещённость пикселя экрана определяется вектором нормали соответствующей вершины. Вектор нормали входит в формулу освещённости. Ранее мы задавали векторы нормалей в вершинах треугольной сетки, а в прочих точках нормаль получалась интерполяцией. Из-за этого было невозможно отобразить детали поверхности меньшие чем размер треугольника в сетке. Техника *рельефного текстурирования* создаёт иллюзию лучшей детализации поверхности за счёт использования текстуры для возмущения нормального вектора в каждой вершине.

5.0.2 Построение рельефного текстурирования

Подробная информация о том, как возмущается нормальный вектор содержится в двухмерном массиве трёхмерных векторов. Массив называется *картой нормалей*. Каждый вектор в карте нормалей показывает направление, в которое должен указывать нормальный вектор относительно интерполированного интерполированного нормального вектора в точке внутри лицевой стороны треугольника. Вектор $\langle 0, 0, 1 \rangle$ — это невозмущённая нормаль. Все прочие вектора представляют собой изменённые нормали.

Обычно карту нормалей получают, извлекая информацию из *карты высот*, которая состоит из высоты рельефа над плоской поверхностью в каждой вершине объекта. Для каждого элемента карты высот вычисляют вектор нормали. При этом сначала вычисляют два касательных в к рельефной поверхности вектора в направлениях s и t , используя разности высот у смежных элементов карты высот. Будем обозначать через $H(i, j)$ высоту элемента с координатами $\langle i, j \rangle$ карты высот размера $w \times h$. Обозначим через $\mathbf{S}(i, j)$ и $\mathbf{T}(i, j)$ касательные вектора в направлениях s и t соответственно:

$$\begin{aligned}\mathbf{S}(i, j) &= \langle 1, 0, aH(i+1, j) - aH(i-1, j) \rangle, \\ \mathbf{T}(i, j) &= \langle 0, 1, aH(i, j+1) - aH(i, j-1) \rangle.\end{aligned}$$

С помощью коэффициента a можно регулировать степень возмущения нормалей. Если обозначить через S_z и T_z z -компоненту векторов $\mathbf{S}(i, j)$ и $\mathbf{T}(i, j)$, то нормальный вектор

$\mathbf{N}(i, j)$ вычисляется как векторное произведение:

$$\mathbf{N}(i, j) = \frac{\mathbf{S}(i, j) \times \mathbf{T}(i, j)}{\|\mathbf{S}(i, j) \times \mathbf{T}(i, j)\|} = \frac{\langle -S_z, -T_z, 1 \rangle}{\sqrt{S_z^2 + T_z^2 + 1}}. \quad (0.1)$$

Компоненты каждого нормального вектора хранятся в виде RGB-цвета. При этом используются следующие формулы:

$$\begin{aligned} \text{Red} &= \frac{x+1}{2}, \\ \text{Green} &= \frac{y+1}{2}, \\ \text{Blue} &= \frac{z+1}{2}. \end{aligned}$$

На рис. 0.1 карта высот в градациях серого и соответствующая карта нормалей, вычисленная с помощью уравнения (0.1).

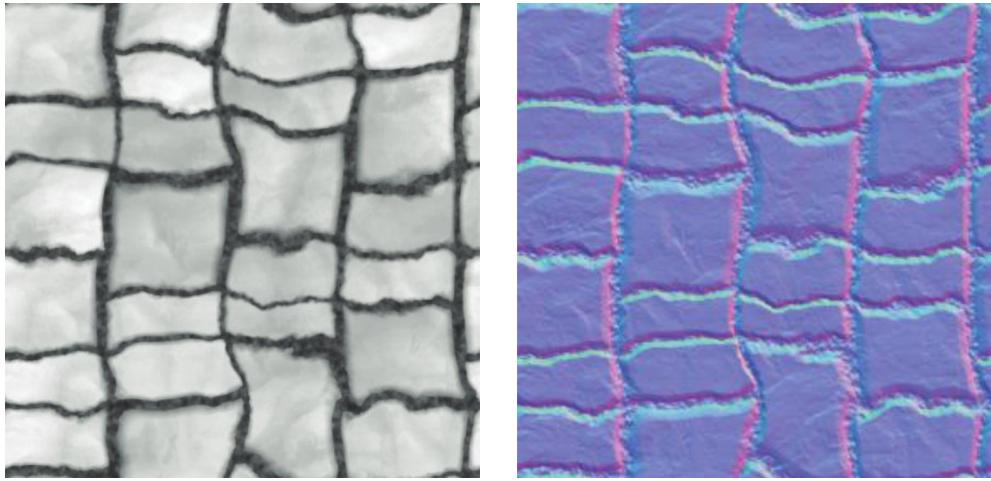


Рис. 0.1: Карта высот и соответствующая ей карта возмущённых нормалей. В карте нормалей преобладает пастельный фиолетовый цвет, так как невозмущённый нормальный вектор $\langle 0, 0, 1 \rangle$ соответствует RGB-цвету $(\frac{1}{2}, \frac{1}{2}, 1)$.

5.0.3 Касательное пространство

Рассмотрим усреднённый (интерполированный) нормальный вектор, используемый в формуле освещения. Поставим ему в соответствие невозмущённый нормальный вектор $\langle 0, 0, 1 \rangle$. Это легко получается, если в каждой вершине построить систему координат так, чтобы вектор нормали всегда совпадал с положительным направлением оси z . Кроме нормального вектора для ортонормированного базиса нам понадобятся ещё два касательных вектора к поверхности в данной вершине. Полученная таким образом система координат называется *касательным пространством* или *пространством вершины* (рис. 0.2).

В каждой вершине треугольной сетки мы имеем касательную систему координат. Направление вектора на источник света \mathbf{L} вычисляется в каждой вершине, а затем определяются его координаты в касательном пространстве. Вектор \mathbf{L} в прилегающем пространстве интерполируется по поверхности треугольника. Так как вектор $\langle 0, 0, 1 \rangle$

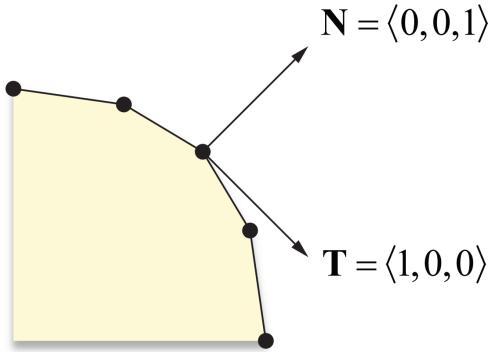


Рис. 0.2: Направление касательного пространства соответствует касательной плоскости и вектору нормали.

в касательном пространстве соответствует вектору нормали, то скалярное произведение вектора направления на источник света и элемента карты нормалей даёт верное значение отражения Ламберта.

Касательные векторы в каждой точке следует выбирать так, чтобы они совпадали направлением координат текстуры. В случае поверхностей построенных на основе параметрических функций касательные вычисляются с помощью частных производных по каждому параметру. В случае произвольной треугольной сетки карта нормалей может применяться в различных направлениях, что требует более общего метода определения направлений касательных в каждой точке.

5.0.4 Вычисление касательных векторов

Найдём для каждой вершины 3×3 -матрицу, которая преобразует пространство сцены в касательное пространство. Для этого вначале решим более простую обратную задачу перехода от касательного пространства к пространству сцены. Так как нормальный вектор к вершине в касательном пространстве соответствует $\langle 0, 0, 1 \rangle$, мы знаем, что ось z касательного пространства всегда сонаравлена нормальному вектору к вершине.

Мы хотим так ориентировать наше касательное пространство, чтобы оси x и y соответствовали s - и t -направлениям карты нормалей. Пусть \mathbf{Q} — это точка внутри треугольника, тогда

$$\mathbf{Q} - \mathbf{P}_0 = (s - s_0)\mathbf{T} + (t - t_0)\mathbf{B},$$

где \mathbf{T} и \mathbf{B} — касательные векторы, \mathbf{P}_0 — одна из вершин треугольника и $\langle s_0, t_0 \rangle$ — координаты текстуры в вершине. Буква \mathbf{B} означает *бикасательный* вектор, направленный перпендикулярно вектору \mathbf{T} .

Предположим треугольник имеет вершины \mathbf{P}_0 , \mathbf{P}_1 и \mathbf{P}_2 . Соответствующие координаты текстуры имеют значения $\langle s_0, t_0 \rangle$, $\langle s_1, t_1 \rangle$ и $\langle s_2, t_2 \rangle$. Для простоты будем вести наши вычисления относительно вершины \mathbf{P}_0 . Пусть

$$\mathbf{Q}_1 = \mathbf{P}_1 - \mathbf{P}_0, \quad \mathbf{Q}_2 = \mathbf{P}_2 - \mathbf{P}_0$$

и

$$\langle s_1, t_1 \rangle = \langle s_1 - s_0, t_1 - t_0 \rangle, \quad \langle s_2, t_2 \rangle = \langle s_2 - s_0, t_2 - t_0 \rangle$$

Нам нужно решить следующие уравнения для \mathbf{T} и \mathbf{B} .

$$\mathbf{Q}_1 = s_1 \mathbf{T} - t_1 \mathbf{B}, \quad \mathbf{Q}_2 = s_2 \mathbf{T} - t_2 \mathbf{B}.$$

Эта система уравнений имеет шесть неизвестных (по три для \mathbf{T} и \mathbf{B}) и шесть уравнений (координаты x , y и z в обоих уравнениях). Перепишем систему в матричном виде

$$\begin{bmatrix} (\mathbf{Q}_1)_x & (\mathbf{Q}_1)_y & (\mathbf{Q}_1)_z \\ (\mathbf{Q}_2)_x & (\mathbf{Q}_2)_y & (\mathbf{Q}_2)_z \end{bmatrix} = \begin{bmatrix} s_1 & t_1 \\ s_2 & t_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}.$$

Умножая обе стороны на обратную к $\langle s, t \rangle$ матрицу, получаем

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} \begin{bmatrix} t_2 & -t_1 \\ -s_2 & s_2 \end{bmatrix} \begin{bmatrix} (\mathbf{Q}_1)_x & (\mathbf{Q}_1)_y & (\mathbf{Q}_1)_z \\ (\mathbf{Q}_2)_x & (\mathbf{Q}_2)_y & (\mathbf{Q}_2)_z \end{bmatrix}.$$

В итоге получает ненормированные касательные вектора \mathbf{T} и \mathbf{B} для треугольника с вершинами \mathbf{P}_0 , \mathbf{P}_1 и \mathbf{P}_2 . Чтобы получить касательный вектор к одной вершине, мы усредняем касательные вектора среди всех треугольников с общей вершиной (аналогично тому, как мы обычно вычисляем вектор нормали). В случае, когда соседние треугольник имеют разрывы текстуры (вершины на границе объекта имеют одинаковые пространственные координаты, но разные текстурные координаты) мы не усредняем касательные, так как результат не будет соответствовать направлению нормали (из карты нормалей).

Имея вектор нормали \mathbf{N} и касательные вектора \mathbf{T} и \mathbf{B} для каждой вершины, мы можем перейти от касательного пространства в пространство сцены с помощью следующей матрицы:

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}.$$

Для обратного преобразования (из пространства сцены в касательное пространство) нам нужно вычислить обратную матрицу. Совсем необязательно, что касательные вектора перпендикулярны друг другу и вектору нормали. Отсюда обратная матрица может не совпадать с транспонированной. Однако можно считать, что эти три вектора почти ортогональны. Значит, используя алгоритм ортогонализации Грама-Шмидта мы не получим неприемлемые результаты. Новые значения векторов \mathbf{T}' и \mathbf{B}'

$$\begin{aligned} \mathbf{T}' &= \mathbf{T} - (\mathbf{N} \cdot \mathbf{T})\mathbf{N}, \\ \mathbf{B}' &= \mathbf{B} - (\mathbf{N} \cdot \mathbf{B})\mathbf{N} - (\mathbf{T}' \cdot \mathbf{B})\mathbf{T}'. \end{aligned}$$

Нормализуем эти векторы и запишем из в виде матрицы перехода от касательного пространства к пространству сцены

$$\begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N'_x & N'_y & N'_z \end{bmatrix}. \tag{0.2}$$

Нет нужды хранить отдельный массив значений координат бикасательного вектора \mathbf{B}' , так как векторное произведение $\mathbf{N} \times \mathbf{T}'$ можно использовать для получения $m\mathbf{B}'$,

где значение $m = \pm 1$ определяет направление касательного пространства. Значение m нужно сохранять для каждой вершины, так как вектор \mathbf{B}' полученный как значение $\mathbf{N} \times \mathbf{T}'$ может быть неверно направлен. Величина m равна определителю матрицы (0.2). Может оказаться удобным хранить вектор \mathbf{T}' как четырёхмерный, чья w -координата содержит значение m . В этом случае вектор \mathbf{B}' равен

$$\mathbf{B}' = T'_w(\mathbf{N} \times \mathbf{T}'),$$

где векторное произведение не учитывает координату w . Получается, что если программа может оперировать с четырёхмерными векторами, то нет необходимости объявлять отдельный массив для значений m .

Ниже представлена программа, вычисляющая касательные вектора для произвольной треугольной сетки. В программе определяются касательный и бикасательный вектора для каждого треугольника в сетке. Эти значения суммируются для каждой вершины, общей для нескольких треугольников. Далее в цикле по всем вершинам касательный и бикасательный вектора ортогоанализируются. В результате получается четырёхмерный касательный вектор с четвёртой координатой, равной значению направления m .

```
void CalculateTangentArray(long vertexCount, const Point3D *vertex,
    const Vector3D *normal, const Point2D *texcoord, long triangleCount,
    const Triangle *triangle, Vector4D *tangent)
{
    Vector3D *tan1 = new Vector3D[vertexCount * 2];
    Vector3D *tan2 = tan1 + vertexCount;
    ZeroMemory(tan1, vertexCount * sizeof(Vector3D) * 2);

    for (long a = 0; a < triangleCount; a++)
    {
        long i1 = triangle->index[0];
        long i2 = triangle->index[1];
        long i3 = triangle->index[2];

        const Point3D& v1 = vertex[i1];
        const Point3D& v2 = vertex[i2];
        const Point3D& v3 = vertex[i3];
        const Point2D& w1 = texcoord[i1];
        const Point2D& w2 = texcoord[i2];
        const Point2D& w3 = texcoord[i3];

        float x1 = v2.x - v1.x;
        float x2 = v3.x - v1.x;
        float y1 = v2.y - v1.y;
        float y2 = v3.y - v1.y;
        float z1 = v2.z - v1.z;
        float z2 = v3.z - v1.z;

        float s1 = w2.x - w1.x;
        float s2 = w3.x - w1.x;
        float t1 = w2.y - w1.y;
        float t2 = w3.y - w1.y;

        float r = 1.0F / (s1 * t2 - s2 * t1);
        Vector3D sdir((t2 * x1 - t1 * x2) * r, (t2 * y1 - t1 * y2) * r,
            (t2 * z1 - t1 * z2) * r);
        Vector3D tdir((s1 * x2 - s2 * x1) * r, (s1 * y2 - s2 * y1) * r,
            (s1 * z2 - s2 * z1) * r);
        tan1[i1] += sdir;
        tan1[i2] += sdir;
        tan1[i3] += sdir;
```

```

        tan2[i1] += tdir;
        tan2[i2] += tdir;
        tan2[i3] += tdir;
        triangle++;
    }

    for (long a = 0; a < vertexCount; a++)
    {
        const Vector3D& n = normal[a];
        const Vector3D& t = tan1[a];

        // Ортогонализация Грама–Шмидта.
        tangent[a] = (t - n * Dot(n, t)).Normalize();

        // Вычислить направление обхода точек треугольника
        // (по часовой или против часовой, handedness).
        tangent[a].w = (Dot(Cross(n, t), tan2[a]) < 0.0F) ? -1.0F : 1.0F;
    }
    delete[] tan1;
}

```

5.0.5 Реализация шейдера

Действия, выполняемые для рельефного текстурирования можно разделить на те, что выполняются для каждой вершины и на те, которые выполняются для каждого пикселя. В каждой вершине нам необходимо определить направление на наблюдателя **V** и направление на источник света **L**, чтобы преобразовать их касательное пространство с помощью (0.2). Представленный ниже вершинный шейдер выполняет эти вычисления для поверхности, освещённой направленным источником света (для которого **L** есть константа).

Вектора **V** и **L** интерполируются по поверхности каждого треугольника. Фрагментный шейдер должен их нормализовать и использовать для вычисления нормализованного промежуточного вектора **H**

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{\|\mathbf{L} + \mathbf{V}\|}.$$

Для каждого фрагмента (каждой вершины) определяются скалярные произведения **N · L** и **N · H**, причём значение **N** берется из карты нормалей. В итоге результаты этих скалярных произведений используются для получения рассеянной и зеркальной составляющих освещённости.

```

in vec4 vertexPosition; // Координаты вершины в объектном пространстве.
in vec3 normal; // Нормаль к вершине в объектном пространстве.
in vec4 tangent; // Направление касательной к вершине в объектном пространстве.
out vec3 view; // Направление глаза наблюдателя в касательном пространстве.
out vec3 light; // Направление света в касательном пространстве.
uniform vec4 mvpMatrix[4]; // Модельно–видовая–проекционная матрица.
uniform vec3 cameraPosition; // Положение наблюдателя в пространстве объекта.
uniform vec3 lightDirection; // Направление света в пространстве объекта.
void main()
{
    // Преобразуем вершину в отсечённое пространство
    gl_Position = vec4(dot(mvpMatrix[0], vertexPosition),
                      dot(mvpMatrix[1], vertexPosition),
                      dot(mvpMatrix[2], vertexPosition),
                      dot(mvpMatrix[3], vertexPosition));
}

```

```
// Вычислим второй касательный вектор  $B = (N \times T) * T.w.$ 
vec3 bitangent = cross(normal, tangent.xyz) * tangent.w;

// Преобразуем  $V$  в касательное пространство.
view = cameraPosition - vertexPosition;
view = vec3(dot(tangent, view), dot(bitangent, view),
dot(normal, view));

// Преобразуем  $L$  в касательное пространство.
light = vec3(dot(tangent, lightDirection),
dot(bitangent, lightDirection), dot(normal, lightDirection));
}
```

Глава 6

Моделирование жидкости и ткани

6.1 Имитация жидкости

6.1.1 Имитация поверхности жидкости

Требуется смоделировать поведение поверхности жидкости так, чтобы сохранялось видимое сходство с настоящими физическими процессами в жидкости. Мы можем добиться этого эффекта, если будем моделировать распространение волны в жидкости.

6.1.2 Уравнение волны

Уравнение волны представляет собой дифференциальное уравнение в частных производных, которое описывает движение каждой точки одномерной пружины или двумерной поверхности, на которые оказывают постоянное воздействие. Чтобы получить одномерное уравнение волны, рассмотрим гибкую эластичную пружину, которая жестко закреплена своими концами и направлено вдоль оси Ox (рис. 1.1). Будем полагать, что пружина имеет постоянную линейную плотность (масса на единицу длины) ρ . На пружину действует постоянная сила T в направлении касательной к пружине.

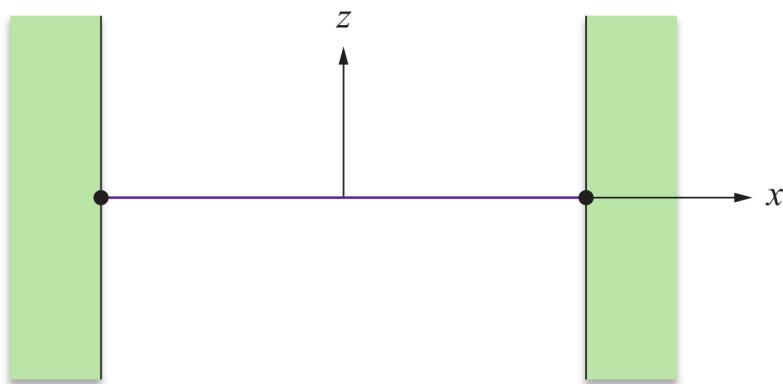


Рис. 1.1: Пружина с линейной плотностью ρ жестко закреплена на своих концах испытывает натяжение T .

Пусть функция $z(x, t)$ означает вертикальное смещение пружины в точке x в момент времени t . Когда пружина смещается в направлении z , натяжение порождает силу в каждой точке вдоль пружины. По второму закону Ньютона сила, действующая на

участок пружины между $x = s$ и $x = s + \Delta x$ в любой момент времени t , равна произведению массы на ускорение $\mathbf{a}(x, t)$. Так как линейная плотность пружины равна ρ , то масса участка равна $\rho\Delta x$, и мы имеем

$$\mathbf{a}(x, t) = \frac{\mathbf{F}(x, t)}{\rho\Delta x}. \quad (1.1)$$

Как видно из рис. 1.2, мы можем разделить силу, действующую на концы участка между $x = s$ и $x = s + \Delta x$, на горизонтальную $H(x, t)$ и вертикальную $V(x, t)$ составляющие. Пусть угол между осью Ox и касательной к пружине в точке $x = s$ равен θ .

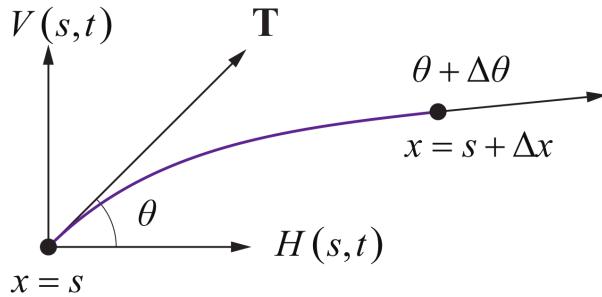


Рис. 1.2: К концам участка с границами $x = s$ и $x = s + \Delta x$ приложены силы, которые можно разложить на вертикальные и горизонтальные составляющие.

Сила \mathbf{T} действует в направлении касательной, отсюда составляющие силы равны

$$\begin{aligned} H(s, t) &= T \cos \theta, \\ V(s, t) &= T \sin \theta. \end{aligned} \quad (1.2)$$

Пусть $\theta + \Delta\theta$ обозначает угол между осью Ox и касательной к пружине в точке $x = s + \Delta x$. Соответствующие составляющие силы равны

$$\begin{aligned} H(s + \Delta x, t) &= T \cos(\theta + \Delta\theta), \\ V(s + \Delta x, t) &= T \sin(\theta + \Delta\theta). \end{aligned}$$

При малых движениях будем считать, что горизонтальная сила равно нулю. Поэтому участок пружины имеет ускорение только в вертикальном направлении. Отсюда потребуем, чтобы для участка $[x; x + \Delta x]$ выполнялось условие

$$H(s + \Delta x, t) - H(s, t) = 0.$$

Видно, что функция H не зависит от x , поэтому можно писать $H(t)$ вместо $H(x, t)$.

Чистая вертикальная сила, действующая на участок $[x; x + \Delta x]$ производит ускорение, которое равное компоненте z в равенстве 1.1. Вертикальное ускорение равно второй производной функции $z(x, t)$

$$a_z(s, t) = \frac{\partial^2}{\partial t^2} z(s, t) = \frac{V(s + \Delta x, t) - V(s, t)}{\rho\Delta x}.$$

Умножим обе части на ρ и устремим Δx к нулю. Получим

$$\rho \frac{\partial^2}{\partial t^2} z(s, t) = \lim_{\Delta x \rightarrow 0} \frac{V(s + \Delta x, t) - V(s, t)}{\Delta x}. \quad (1.3)$$

Правая часть (1.3) равна по определению частной производной V по x в при $x = s$. Следовательно,

$$\rho \frac{\partial^2}{\partial t^2} z(s, t) = \frac{\partial}{\partial x} V(s, t). \quad (1.4)$$

Используя значения $H(t)$ и $V(s, t)$ из уравнения (1.2), свяжем компоненты воедино

$$V(s, t) = H(t) \operatorname{tg} \theta.$$

Очевидно, $\operatorname{tg} \theta$ совпадает с производной функции $z(x, t)$ по x при $x = s$

$$V(s, t) = H(t) \frac{\partial}{\partial x} z(s, t).$$

Уравнение (1.4) принимает вид

$$\rho \frac{\partial^2}{\partial t^2} z(s, t) = \frac{\partial}{\partial x} \left[H(t) \frac{\partial}{\partial x} z(s, t) \right].$$

Так как $H(t)$ не зависит от x , можно записать

$$\rho \frac{\partial^2}{\partial t^2} z(s, t) = H(t) \frac{\partial^2}{\partial x^2} z(s, t).$$

При малых смещениях $\cos \theta$ близок к 1, поэтому приближённо можно заменить $H(t)$ на натяжение $T(x)$. Если положить $c^2 = T/\rho$, мы придём к уравнению одномерной волны:

$$\frac{\partial^2 z}{\partial t^2} = c^2 \frac{\partial^2 z}{\partial x^2}. \quad (1.5)$$

Уравнение для двумерной волны получается добавлением второго пространственного слагаемого в уравнении (1.5)

$$\frac{\partial^2 z}{\partial t^2} = c^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right). \quad (1.6)$$

Постоянная c имеет размерность расстояния на единицу времени, поэтому представляет собой скорость. Примем без доказательства, что c — это скорость распространения волны вдоль пружины или по поверхности. Очевидно, что скорость распространения волны возрастает с увеличением натяжения, возникающего внутри среды (пружины или поверхности), и уменьшается с ростом плотности среды.

Уравнение (1.6) не учитывает никакие силы кроме натяжения поверхности. Поэтому средняя амплитуда волны на поверхности никогда не уменьшается по сравнению с действительностью. Мы можем ввести в уравнение силу вязкого замедления, добавив силу, действующую в обратном направлении к скорости точки на поверхности

$$\frac{\partial^2 z}{\partial t^2} = c^2 \left(\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right) - \mu \frac{\partial z}{\partial t}, \quad (1.7)$$

где неотрицательная постоянная μ является вязкостью жидкости. Величина μ определяет, за какое время исчезают волны на поверхности. Малое значение μ позволяет волнам долго сохраняться, как в случае воды. Большое значение μ быстро уменьшает волны, как в случае вязкого масла.

6.1.3 Расчёт смещения поверхности

Составим для дифференциального уравнения (1.7) следующее сеточное уравнение

$$\begin{aligned} \frac{z_{i,j}^{k-1} - 2z_{i,j}^k + z_{i,j}^{k+1}}{\tau^2} &= \\ &= c^2 \frac{z_{i-1,j}^k - 2z_{i,j}^k + z_{i+1,j}^k}{d^2} + c^2 \frac{z_{i,j-1}^k - 2z_{i,j}^k + z_{i,j+1}^k}{d^2} - \mu \frac{z_{i,j+1}^k - z_{i,j-1}^k}{2t}. \end{aligned} \quad (1.8)$$

Очевидно, погрешность аппроксимации равна $O(d^2 + \tau^2)$. Мы хотим определять будущие смещения поверхности $z_{i,j}^{k+1}$ спустя промежуток времени τ , прошедший от настоящего момента. Считаем, что текущее $z_{i,j}^k$ и предыдущее $z_{i,j}^{k-1}$ смещения нам известно. Решая уравнение (1.8) относительно $z_{i,j}^{k+1}$, получим

$$\begin{aligned} z_{i,j}^{k+1} &= \frac{4 - 8c^2\tau^2/d^2}{\mu\tau + 2} z_{i,j}^k + \frac{\mu\tau - 2}{\mu\tau + 2} z_{i,j}^{k-1} + \\ &\quad + \frac{2c^2\tau^2/d^2}{\mu\tau + 2} [z_{i+1,j}^k + z_{i-1,j}^k + z_{i,j+1}^k + z_{i,j-1}^k]. \end{aligned} \quad (1.9)$$

Если постоянные множители в каждом слагаемом можно вычислить заранее, то на каждую вершину сетки потребуется 3 операции умножения и 4 операции сложения.

Если скорость волны с слишком велика, или шаг по времени τ очень большой, то последовательное применение равенства (1.9) уводит смещения вершин в бесконечность (разностная схема неустойчива). Чтобы смещения были конечными, нужно выяснить условия устойчивости (1.9). Условия вытекают из требования, чтобы всякая вершина, которую сместили относительно оставшейся плоской поверхности, должна стремиться в сторону поверхности, когда на вершину перестанут воздействовать.

Пусть дан массив вершин размера $m \times n$, причём $z_{i,j}^0 = z_{i,j}^1 = 0$ для всех вершин кроме имеющих координаты (i_0, j_0) . Пусть вершина (i_0, j_0) смещается таким образом, что $z_{i_0,j_0}^0 = z_{i_0,j_0}^1 = h$, где h — это ненулевое расстояние. Теперь предположим, что в момент 2τ вершину (i_0, j_0) отпускают. При расчёте значения z_{i_0,j_0}^2 третий член в 1.9 равен нулю, поэтому

$$z_{i_0,j_0}^2 = \frac{4 - 8c^2\tau^2/d^2}{\mu\tau + 2} z_{i_0,j_0}^1 + \frac{\mu\tau - 2}{\mu\tau + 2} z_{i_0,j_0}^0 = \frac{2 - 8c^2\tau^2/d^2 + \mu\tau}{\mu\tau + 2} h. \quad (1.10)$$

Считаем (требуем), что частица, возвращающаяся в своё исходное положение в плоской поверхности, через промежуток времени 2τ будет ближе к поверхности, чем через τ . Запишем это условие

$$|z_{i_0,j_0}^2| < |z_{i_0,j_0}^1| = |h|.$$

Подставим значение z_{i_0,j_0}^2 , взятое из (1.10)

$$\left| \frac{2 - 8c^2\tau^2/d^2 + \mu\tau}{\mu\tau + 2} \right| |h| < |h|.$$

Таким образом,

$$-1 < \frac{2 - 8c^2\tau^2/d^2 + \mu\tau}{\mu\tau + 2} < 1. \quad (1.11)$$

Решая относительно c , получим

$$0 < c < \frac{d}{2\tau} \sqrt{\mu\tau + 2}. \quad (1.12)$$

Отсюда следует, что для любого расстояния d между соседними вершинами и любого промежутка времени τ скорость c должна быть меньше наибольшего значения в правой части неравенства (1.12).

С другой стороны, имея значения расстояния d и скорости c , мы можем определить максимальное значение для промежутка времени τ . Умножая обе части (1.11) на $-(\mu t + 2)$ и упрощая, получим

$$0 < \frac{4c^2}{d^2}\tau^2 < \mu\tau + 2.$$

Левое неравенство требует, чтобы $\tau > 0$. Но это и так есть естественное предположение о промежутке времени. Правое неравенство даёт квадратичное выражение

$$\frac{4c^2}{d^2}\tau^2 - \mu\tau - 2 < 0. \quad (1.13)$$

Корни квадратного уравнения

$$\tau_{1,2} = \frac{\mu \pm \sqrt{\mu^2 + 32c^2/d^2}}{8c^2/d^2}. \quad (1.14)$$

Так как коэффициент при главном члене в (1.13) положительный, то соответствующая парабола вогнута вверх, а значение многочлена будет отрицательным при $\tau_2 < \tau < \tau_1$. Запишем теперь ограничение на интервал времени τ

$$0 < \tau < \frac{\mu + \sqrt{\mu^2 + 32c^2/d^2}}{8c^2/d^2}. \quad (1.15)$$

Применение значений скорости распространения волны c за пределами области (1.12) или значений τ за пределами области (1.15) приводит к экспоненциальному росту смещения вершин поверхности.

6.1.4 Реализация

При решении уравнения (1.9) для поверхности жидкости требуется хранить два буфера, содержащих по $n \times m$ положений вершин. При отрисовке очередного кадра один из буферов содержит текущие положения вершин, а второй буфер содержит предыдущие положения вершин. Рассчитывая смещения вершин для очередного кадра, мы заменяем вершины в буфере со старыми положениями вершин на новые положения вершин. Буфер с текущими положениями превращается в буфер со старыми значениями. Таким образом, каждый раз при отрисовке очередного кадра мы меняем ролями буфера.

Чтобы правильно рассчитать освещённость, нужно знать истинный вектор нормали для каждой вершины. Для вершины с координатами (i, j) ненормализованные каса-

тельные вдоль оси Ox вектор \mathbf{T} и вдоль оси Oy вектор \mathbf{B} получаются как

$$\begin{aligned}\mathbf{T} &= \left\langle 1, 0, \frac{\partial}{\partial x} z(i, j, k) \right\rangle \\ \mathbf{B} &= \left\langle 0, 1, \frac{\partial}{\partial y} z(i, j, k) \right\rangle.\end{aligned}$$

Заменяя частные производные разделёнными разностями, получим

$$\begin{aligned}\mathbf{T} &= \left\langle 1, 0, \frac{z_{i+1,j}^k - z_{i-1,j}^k}{2d} \right\rangle \\ \mathbf{B} &= \left\langle 0, 1, \frac{z_{i,j+1}^k - z_{i,j-1}^k}{2d} \right\rangle.\end{aligned}$$

Ненормализованный вектор нормали \mathbf{N} получается из векторного произведения $\mathbf{N} = \mathbf{T} \times \mathbf{B}$:

$$\mathbf{N} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 1 & 0 & \frac{z_{i+1,j}^k - z_{i-1,j}^k}{2d} \\ 0 & 1 & \frac{z_{i,j+1}^k - z_{i,j-1}^k}{2d} \end{vmatrix} = \left\langle -\frac{z_{i+1,j}^k - z_{i-1,j}^k}{2d}, -\frac{z_{i,j+1}^k - z_{i,j-1}^k}{2d}, 1 \right\rangle.$$

Умножим вектора \mathbf{T} , \mathbf{B} и \mathbf{N} на $2d$. От этого направления векторов не изменятся, но исчезнет операция деления. Получим

$$\begin{aligned}\mathbf{T} &= \langle 2d, 0, z_{i+1,j}^k - z_{i-1,j}^k \rangle \\ \mathbf{B} &= \langle 0, 2d, z_{i,j+1}^k - z_{i,j-1}^k \rangle \\ \mathbf{N} &= \langle z_{i-1,j}^k - z_{i+1,j}^k, z_{i,j-1}^k - z_{i,j+1}^k, 2d \rangle.\end{aligned}$$

Ниже приведён пример программы, где реализована имитация поверхности жидкости. Важно понимать, что шаг по времени в разностной схеме является константой. При анимации может меняться число отображаемых кадров в единицу времени. Поэтому очень важно убедиться перед отрисовкой поверхности жидкости, что прошло достаточно количества времени. Особенно это важно при большом числе кадров в единицу времени.

Если с поверхностью жидкости взаимодействует некоторый объект (например, в воду брошен камень), то появляются «воздушения». Поверхность можно «подправить», явно меняя текущее и предыдущее положения вершин, окружающих точку, где произошло взаимодействие. Замещение вершины близкой к точке удара и меньшим количеством. Восемь ближайших соседей в общем случае даёт хорошие результаты.

В программе реализован алгоритм, строящий поверхность по двум буферам. В конструктор класса `Fluid` передаётся размер массива вершин, расстояние d между соседними вершинами, шаг по времени τ , скорость волны c и вязкость μ . Поле `renderBuffer` показывает, какой буфер отображается в текущий момент времени, значение поля циклически переключается между 0 и 1 при каждом вызове метода `Fluid::Evaluate()`.

```
class Fluid
{

```

```

private:
    long width;
    long height;

    Vector3D *buffer[2];
    long renderBuffer;

    Vector3D *normal;
    Vector3D *tangent;
    float k1, k2, k3;

public:
    Fluid(long n, long m, float d, float t, float c, float mu);
    ~Fluid();

    void Evaluate(void);
};

Fluid::Fluid(long n, long m, float d, float t, float c, float mu)
{
    width = n;
    height = m;
    long count = n * m;

    buffer[0] = newVector3D[count];
    buffer[1] = newVector3D[count];
    renderBuffer = 0;

    normal = newVector3D[count];
    tangent = newVector3D[count];

    // Подготовим заранее константы для уравнения (1.9).
    float f1 = c * c * t * t / (d * d);
    float f2 = 1.0F / (mu * t + 2);
    k1 = (4.0F - 8.0F * f1) * f2;
    k2 = (mu * t - 2) * f2;
    k3 = 2.0F * f1 * f2;

    // Начальные значения буферов.
    long a = 0;
    for(long j = 0; j < m; j++)
    {
        float y = d * j;
        for(long i = 0; i < n; i++)
        {
            buffer[0][a].Set(d * i, y, 0.0F);
            buffer[1][a] = buffer[0][a];
            normal[a].Set(0.0F, 0.0F, 2.0F * d);
            tangent[a].Set(2.0F * d, 0.0F, 0.0F);
            a++;
        }
    }
}

Fluid::~Fluid()
{
    delete[] tangent;
    delete[] normal;
    delete[] buffer[1];
    delete[] buffer[0];
}

void Fluid::Evaluate(void)
{
    // Применим уравнение (1.9).
    for(long j = 1; j < height - 1; j++)
    {
        const Vector3D *crnt = buffer[renderBuffer] + j * width;
        Vector3D *prev = buffer[1 - renderBuffer] + j * width;
        for(long i = 1; i < width - 1; i++)
        {

```

```

    prev[i].z = k1 * crnt[i].z + k2 * prev[i].z +
    k3 * (crnt[i + 1].z + crnt[i - 1].z +
    crnt[i + width].z + crnt[i - width].z);
}
// Поменяем местами буферы.
renderBuffer = 1 - renderBuffer;

// Вычисляем нормали и касательные.
for(long j = 1; j < height - 1; j++)
{
    constVector3D *next = buffer[renderBuffer] + j * width;
    Vector3D *nrml = normal + j * width;
    Vector3D *tang = tangent + j * width;

    for(long i = 1; i < width - 1; i++)
    {
        nrml[i].x = next[i - 1].z - next[i + 1].z;
        nrml[i].y = next[i - width].z - next[i + width].z;
        tang[i].z = next[i + 1].z - next[i - 1].z;
    }
}
}

```

6.2 Имитация ткани

Примером имитации ткани может служить флаг, развивающийся на ветру. Истинные физические процессы, лежащие в основе этого движения, оказываются слишком сложными. Но если моделировать ткань прямоугольными кусочками, то программная реализация движения ткани будет достаточно простой, а результаты реалистичными. Рассмотрим далее основные этапы имитации движения ткани.

6.2.1 Система пружин

Ткань представляет собой двумерную регулярную сетку точечных частичек. Мы будем имитировать движение только этих частичек. Потом объединим вершины сетки в треугольные примитивы, чтобы вывести изображение ткани на экран. Будем считать ткань упругой и неразрывной. Чтобы контролировать её движение, будем считать, что каждая частичка соединена с несколькими соседними воображаемыми пружинами. Каждая пружина сохраняет определённое расстояние между частицами и препятствует тому, чтобы соседние частицы имели слишком отличающиеся скорости.

На рис. 2.3 представлены связи для отдельной частицы внутри сетки, имитирующей ткань. Во-первых, пружины соединяют частицу со своими ближайшими соседями слева, справа, сверху и снизу, как показано красными линиями на рисунке. Эти связи определяют основную структуру ткани. Они также не позволяют частицам разлетаться или совмещаться в одной точке. Однако они не препятствуют нежелательному очень грубому загибу ткани. Чтобы ограничить степеньгиба, пружины соединяют частицу с соседями на расстоянии двух вершин слева, справа, сверху и снизу, как показано зелёными линиями на рисунке. За счёт этих восьми связей модель ткани ведёт себя очень реалистично. Остаётся только устраниТЬ так называемую деформацию сдвига. Дело в том, что расстояния между восемью упомянутыми соседями остаётся неизмен-

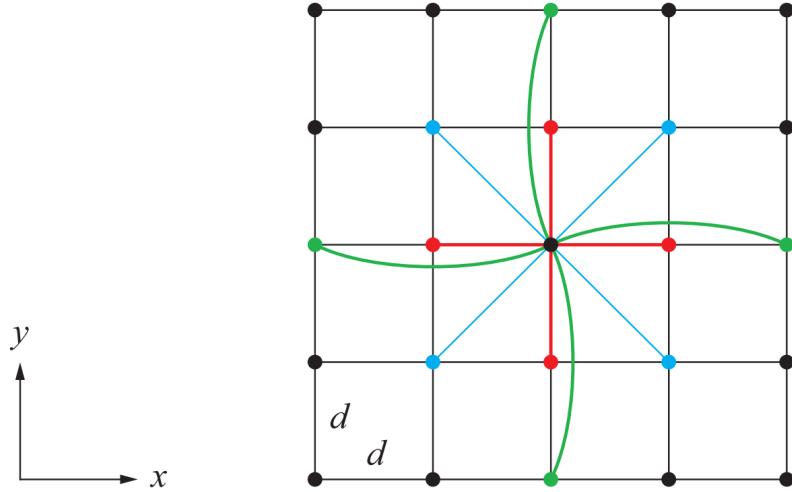


Рис. 2.3: Центральная частица соединена с четырьмя ближайшими соседями (красные линии), чтобы передать ткани неразрывность, с четырьмя соседями на расстоянии двух вершин (зелёные линии), чтобы ограничить изгиб, и с четырьмя соседями по диагонали (голубые линии), чтобы предотвратить деформацию сдвига.

ным даже, если деформировать сетку, превратив квадраты в ромбы. Чтобы исключить этот случай, добавим четыре пружины между частицей и её соседями по диагоналям, как показано голубыми линиями на рисунке. Для частиц на границе сетки количество пружин будет меньше, так как не все соседние частицы могут существовать.

Пусть \mathbf{P} и \mathbf{Q} обозначают положения двух ближайших соседних частиц, связанных пружиной, и пусть d — это длина пружины в состоянии покоя. Пружина стремиться сохранить между двумя частицами расстояние d . Если же расстояние между частицами отличается от d , то на частицы действует сила пружины $\mathbf{F}_{\text{пружина}}$, равная

$$\mathbf{F}_{\text{пружина}} = k_{\text{пружина}} (\|\mathbf{Q} - \mathbf{P}\| - d) \frac{\mathbf{Q} - \mathbf{P}}{\|\mathbf{Q} - \mathbf{P}\|}, \quad (2.16)$$

где $k_{\text{пружина}}$ — постоянный коэффициент пружины. На частицу \mathbf{P} действует сила $\mathbf{F}_{\text{пружина}}$, а на частицу \mathbf{Q} действует сила $-\mathbf{F}_{\text{пружина}}$. Для пружин, соединяющих соседей на расстоянии двух вершин и соседей по диагоналям, длины покоя будут соответственно $2d$ и $\sqrt{2}d$.

Действие каждой пружины зависит также от скорости движения частиц. Сила $\mathbf{F}_{\text{амортизатор}}$ возникает, если имеется разница в скоростях $d\mathbf{P}/dt$ и $d\mathbf{Q}/dt$. Она равна

$$\mathbf{F}_{\text{амортизатор}} = k_{\text{амортизатор}} \left(\frac{d\mathbf{Q}}{dt} - \frac{d\mathbf{P}}{dt} \right),$$

где $k_{\text{амортизатор}}$ — это положительная константа, определяющая степень амортизации. Как и ранее на частицу $\mathbf{F}_{\text{амортизатор}}$, а на частицу \mathbf{Q} действует сила $-\mathbf{F}_{\text{амортизатор}}$.

Для придания нашей модели ткани большего реализма, можно выбрать различные значения $k_{\text{пружина}}$ и $k_{\text{амортизатор}}$ для пружин, соединяющих различные типы соседей. Например, можно использовать большое значение $k_{\text{пружина}}$ для длинных пружин (соединяющих соседей на расстоянии двух вершин), чтобы придать ткани большую со-

противляемость к сгибу. Очень важно не выбирать слишком большое значение для $k_{\text{пружина}}$, так как очень упругие пружины порождают нежелательные вибрации.

6.2.2 Внешние силы

Если все частицы ткани в начальный момент времени находятся в таких положениях, что все пружины имеют в точности длину покоя, то ничего никакого движения не будет. Чтобы увидеть движение ткани во времени, нужно приложить внешние силы. Самой естественной силой будет сила тяжести. На каждую частицу ткани действует сила

$$\mathbf{F}_{\text{тяж.}} = m\mathbf{g},$$

где m — это масса частицы, и \mathbf{g} — ускорение свободного падения (будем считать, что $\mathbf{g} = \langle 0; 0; -9,8 \frac{\text{м}}{\text{с}^2} \rangle$).

Следующая сила, часто действующая на ткань, порождается ветром или, более точно, разницей между скоростью движения ткани и скоростью движения воздуха. Сила, порождаемая ветром, действующая на частицу ткани \mathbf{P} , где единичный вектор нормали к ткани равен \mathbf{N} , равна

$$\mathbf{F}_{\text{ветер}} = k_{\text{ветер}} \left| \left(\mathbf{W} - \frac{d\mathbf{P}}{dt} \right) \cdot \mathbf{N} \right|.$$

Здесь \mathbf{W} обозначает скорость ветра, а $k_{\text{ветер}}$ — это константа, определяющая, как быстро ветер ускоряет ткань. Параметр $k_{\text{ветер}}$ может принимать разные значения в зависимости от плотности воздуха или от того, может ли воздух проходить сквозь ткань. Благодаря скалярному произведению с вектором нормали сила будет наибольшей, когда направление движения ветра перпендикулярно касательной к ткани плоскости. Сила будет наименьшей, когда ткань расположена вдоль направления ветра. Мы бе-рём абсолютную величину, так как неважно, какая сторона ткани обдувается ветром.

6.2.3 Реализация

Для каждой частицы ткани мы храним положение в трёхмерном пространстве и трёхмерный вектор скорости. Ткань приходит в движение на экране за счёт изменения положений и скоростей всех частиц в течение фиксированного временного интервала Δt . Этот промежуток времени не зависит от частоты смены кадров, с которой сцена рисуется на экране. Таким образом, шагов для имитации движения ткани за интервал рисования сцены на экране может быть различным. Интервал времени от 5 до 20 мс обычно даёт хорошие результаты.

На каждом шаге новое положение частицы вычисляется на основании текущей скорости и совокупности сил, приложенных к частице. Когда новое положение частицы определено, мы рассчитываем новый вектор скорости (который понадобится в следующем шаге), деля разницу между новым и старым положениями на интервал Δt .

Мы можем перебрать все частицы и рассчитать внешние силы, действующие на каждую вершину в отдельности. Для i -ой частицы, чьё положение есть \mathbf{P}_i , чья нормаль

к поверхности есть \mathbf{N}_i и чья текущая скорость есть \mathbf{V}_i , внешняя сила $\mathbf{F}_{\text{внеш.}}$ будет равна

$$\mathbf{F}_{\text{внеш.}} = m\mathbf{g} + k_{\text{внеш.}}|(\mathbf{W} - \mathbf{V}_i) \cdot \mathbf{N}_i|. \quad (2.17)$$

Если мы будем вычислять $\mathbf{F}_{\text{пружина}}$ и $\mathbf{F}_{\text{амортизатор}}$ для каждой вершины в отдельности, то выполним двойную работу, так как на концах любой пружины упомянутые силы одинаковы по модулю и отличаются направлением. Кроме того не все частицы имеют одинаковое число связей (пружин) с прочими частицами. От этого возникают дополнительные сложности при написании программы. Более привлекательный подход состоит в следующем. Мы заполняем массив сил начальными значениями, по одному значению на частицы в соответствии с уравнением (2.17). Далее в цикле проходим по всем пружинам, вычисляя силы для каждой пружины, а затем применяем эти силы к вершинам на концах пружины. Сила, с которой пружина действует на одну частицу, равна силе, действующей на другую частицу с обратным знаком.

Когда суммарная сила \mathbf{F}_i , действующая на частицу \mathbf{P}_i определена, мы вычисляем новое положение \mathbf{P}'_i частицы

$$\mathbf{P}'_i = \mathbf{P}_i + \mathbf{V}_i \Delta t + \frac{\mathbf{F}_i}{2m} (\Delta t)^2.$$

Когда получены новые положения для всех частиц, мы рассчитываем нормальный и касательный вектора. Эти вектора понадобятся для правильной отрисовки поверхности, а кроме того для уравнения (2.17) на следующем шаге по времени.

6.3 Упражнения

- Пусть поверхность жидкости моделируется сеткой и значения координат вершин вычисляются с частотой 20 раз в секунду. Расстояние между соседними вершинами 0,1 м, а вязкость жидкости равна $\mu = 1 \text{ c}^{-1}$. При каком наибольшем значении скорости волны с уравнение (1.9) будет устойчиво?
- Пусть расстояние между соседними вершинами поверхности жидкости равно 0,1 м, а вязкость жидкости равна $\mu = 1 \text{ c}^{-1}$. Какой наибольший промежуток времени τ между соседними итерациями гарантирует устойчивость численного решения (1.9) при скорости волны 2 м/с?

Глава 7

Лабораторный практикум

7.1 Лабораторная работа 1. Знакомство с шейдерами.

Упражнение. С помощью фрагментного шейдера сделать цветное изображение чёрно-белым.

Основное задание. Реализовать с помощью шейдеров модель Фонга.

7.2 Лабораторная работа 2. Рельефное текстурирование.

1. Выбрать текстуру в градациях серого. Это будет карта высот. Белый цвет — это высокие участки, а темный — низкие.
2. Переходим от карты высот к карте нормалей и переводим последнюю в формат текстуры.
3. Накладываем на объект на сцене карту нормалей как текстуру.
4. В шейдерах извлекаем из текстуры нормаль.
5. С помощью модели Фонга отображаем объект с рельефом.

Задание имеет два уровня сложности:

1. Наложение рельефа на плоский объект.
2. Наложение рельефа на неплоский объект.

7.3 Лабораторная работа 3. Эффект тени.

1. Помещаем временно камеру в позицию источника света. На дисплее ничего еще не видно.
2. Делаем снимок из z-буфера и упаковываем его в текстуру.
3. В шейдере сравниваем расстояние от источника света до каждого пикселя на экране. Если расстояние больше, чем значение из текстуры z-буфера, то данный пиксель находится в области тени, иначе пиксель освещён напрямую источником

света. Если пиксель в области тени, то уменьшаем его яркость, иначе выводим полную яркость.

7.4 Лабораторная работа 4. Математические модели объектов.

Задание имеет два варианта:

1. Имитация поверхности жидкости.
2. Имитация поверхности ткани.

7.5 Лабораторная работа 5. Работа с физическим процессором.

Задание основано на библиотеке Box2D. Предлагается выбрать произвольно сцену с несколькими объектами. Необходимо реализовать следующее: качение, связи между объектами.

Литература

- [1] *Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane OpenGL programming guide : the official guide to learning OpenGL, version 4.3* // the Khronos OpenGL ARB Working Group.
- [2] *David Wolff OpenGL 4.0 Shading Language Cookbook* // Birmingham-Mumbai, Packt Publishing.
- [3] *Eric Lengyel Mathematics for 3D Game Programming and Computer Graphics* // Course Technology CENGAGE Learning.
- [4] Мальцев Д. А. Мультимедиа Технологии. Основы работы с редактором шейдеров FX Composer : учебно-методическое пособие // Ульяновск : УлГУ, 2011. – 46 с.
<http://staff.ulstu.ru/maltsevda/fls/ms02.pdf>

