Программирование графических приложений

Тема 17
Разработка игровой анимированой сцены 1

Цель изучения темы. Изучение методов создания игровой анимированной 3D-сцены в WebGL на JavaScript с использованием библиотеки Three.js.

В качестве примера будет рассмотрено построение анимированной сцены – игры пинг-понг с управлением и игровыми функциями. Сцена будет представлять собой игоровой стол, две ракетки (дощечки) и шарик, движущийся по столу между дощечками (рис. 17.1).

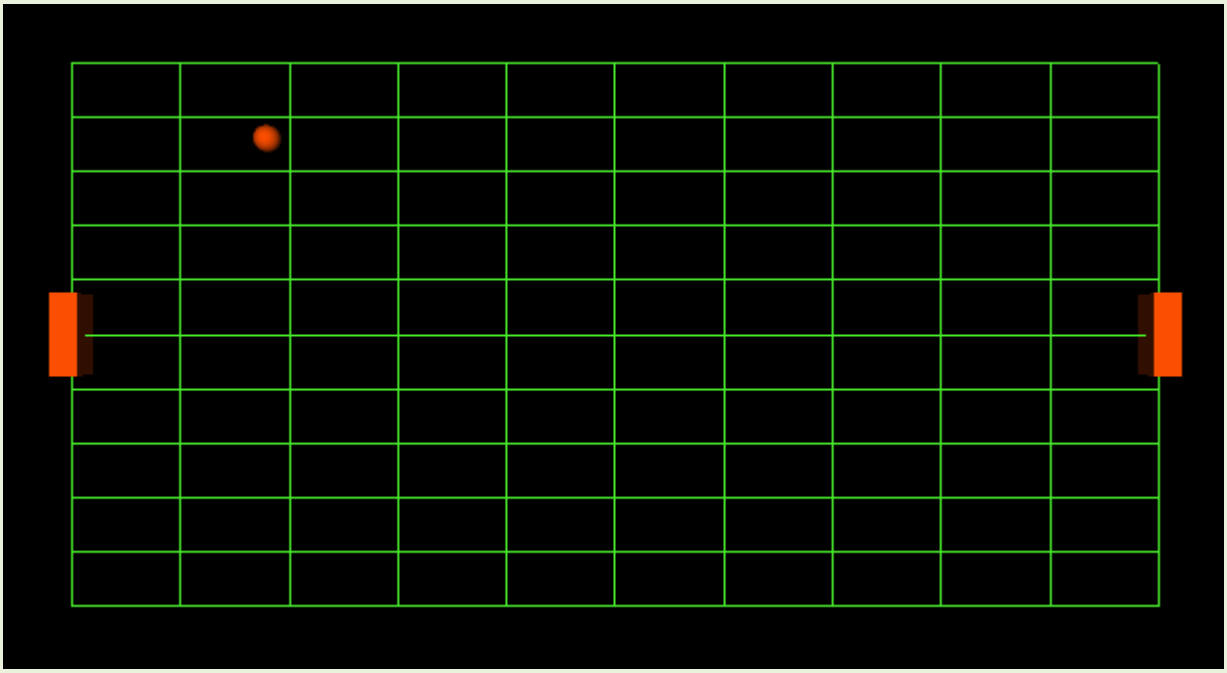

Рис. 17.1

При реализации браузерной 3D игры кроме библиотеки Three.js будет использована библиотека Keyboard.js для управления объектами с помошью клавиатуры

17.1. Подготовка

Подготовка основного файла

Для начала создадим HTML-файл и назовем его ex17_01.html. Состоять он будет всего из нескольких тегов, поэтому CSS стили не будут выноситься в отдельный файл, а пропишатся сразу в теге head.

```
<!doctype html>
<html>
<head>
<title>Пинг-понг</title>
<style>
 body {
 background: #000000;
        }
 #gameCanvas {
 background: #000000;
 width: 640px;
 height: 360px;
 margin: auto;
 align: center;
```

```
            }
 #scoreboard {
 text-align: center;
 font-family: Helvetica, sans-serif;
 color: white;
              }
 #scores {
 font-size:600%;
 padding:0;
 margin:0;
 color: #ffffff;
        }
 #title {
 background: #ffffff;
 color: #000000;
       }
</style>
</head>
<body onload='setup();'>
<div id='gameCanvas'></div>
<script src='./js/three.min.js'></script>
<script src='./js/keyboard.js'></script>
<script src='./js/ex17_01.js'></script>
<div id='scoreboard'>
<h1 id='scores'>0-0</h1>
<h1 id='title'>3D ПИНГ-ПОНГ</h1>
<h2 id='winnerBoard'>Набравший 7 очков победит!</h2>
<h3>A - перемещение влево
<br>D - перемещение вправо</h3>
</div>
</body>
</html>
```

Создаём функции setup() и draw()

Мы создадим две функции setup() и draw(). Функция setup() предназначена для самой первой сцены нашей игры. Функция draw() будет запускаться на каждом кадре и будет управлять рендерингом и логикой игры.

```
function setup() {
    draw();
}

 function draw() {
    requestAnimationFrame(draw);
}
```

Для того, чтобы зациклить функцию draw(), мы вызовем функцию requestAnimationFrame() и передадим ей параметр draw. Так как requestAnimationFrame() не гарантирует постоянную частоту кадров, иногда приходится использовать временные дельты чтобы анимация выглядела более реалистично. Однако для такой простой игры как пинг-понг это не потребуется.

17.2. Формирование сцены

Используем библиотеку Three.js

Библиотека Three.js позволяет создать несколько важных элементов игры:

- Сцена
- Рендер (отвечает за отрисовку объектов)
- Камера
- Меш (объект, состоящий из треугольников, с наложенной на них текстурой)
- Свет
- Материал

Камера, меш и свет добавляются в сцену с помощью функции scene.add().

Прикрепляем рендер к блоку div

Рендер может прикрепляться к любому DOM-элементу HTML, в котором вы хотите создать сцену. После этого в каждом кадре вызывается функция render() для отрисовки сцены.

```
// определяем размер сцены
var WIDTH = 640,
HEIGHT = 360;

// создаем WebGL рендер
var renderer = new THREE.WebGLRenderer();

// запуск рендера
renderer.setSize(WIDTH, HEIGHT);

// прикрепляем блок с id=gameCanvas к рендеру
var c = document.getElementById("gameCanvas");
c.appendChild(renderer.domElement);

...

function draw()
{
// отрисовываем THREE.JS сцену
renderer.render(scene, camera);

// зацикливаем функцию draw()
requestAnimationFrame(draw);

// обработка игровой логики
...
}
```

Добавляем камеру на сцену

С помощью Three.js можно создавать два вида камер: перспективную и ортогональную. Для большинства задач предпочтительной является перспективная камера. Как и любой объект сцены, мы можем менять положение и поворот камеры.

```
camera = new THREE.PerspectiveCamera(
VIEW_ANGLE,
ASPECT,
NEAR,
```

```
FAR);

scene = new THREE.Scene();

// добавляем камеру на сцену
scene.add(camera);

// устанавливаем начальную позицию камеры
// если этого не сделать, то может
// испортиться рендеринг теней
camera.position.z = 320;
```

17.3. Добавляем объекты игры

Рисуем сферу и подсвечиваем ее

Меши нужно использовать вместе с материалами, чтобы придать им нужный внешний вид. В зависимости от примитивов, из которых они состоят (куб, сфера, плоскость или тор), меши выглядят по-разному. Материалы мешей могут иметь различные характеристики в зависимости от их типа; чаще всего это Lambert, Phong и Basic. Напомним:

Basic - рендерит неосвещенный меш без теней и затемнений; при таком материале сфера выглядит обычным кругом;

Lambert - материал с простым диффузным освещением, которое затемняет области объекта, удаленные от источника света, что придает объекту с матовой (не блестящей и не отражающей) поверхностью объемность.

Phong - материал, который отражает падающий свет от источников освещения и отраженный свет от других объектов.

С помощью приведенного ниже кода, создаем сферу с материалом Lambert (рис. 17.2):

```
// устанавливаем переменные для
// сферы: radius, segments, rings
// маленькие значения 'segment' и 'ring'
// улучшают производительность
var radius = 5,
segments = 6,
rings = 6;

// создаем материал сферы
var sphereMaterial =
new THREE.MeshLambertMaterial(
{
color: 0xD43001
});

// создаем шар с геометрией как у сферы
var ball = new THREE.Mesh(
new THREE.SphereGeometry(radius,
segments,
rings),
sphereMaterial);

// добавляем сферу на сцену
scene.add(ball);
```

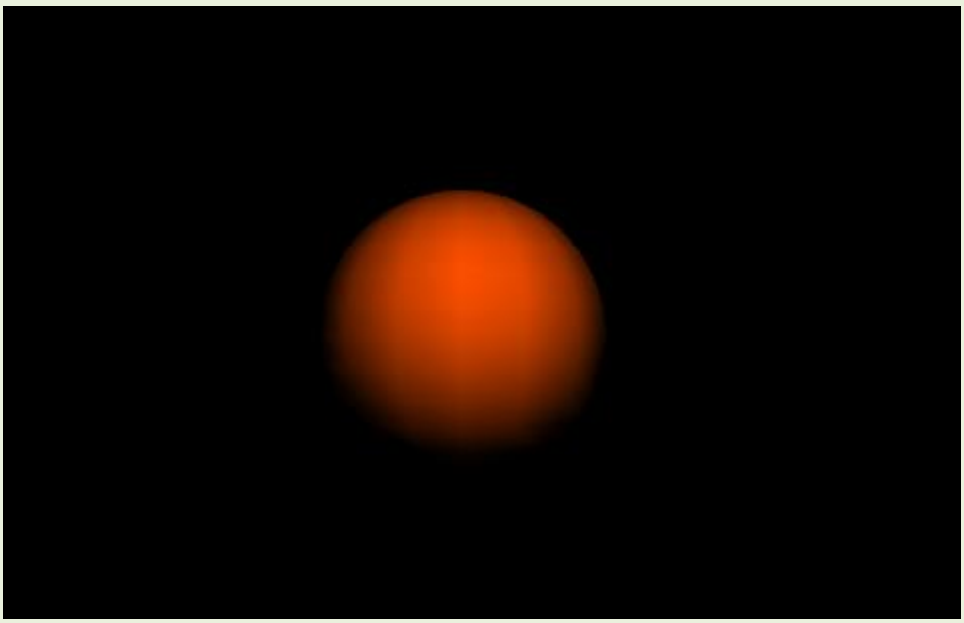

Рис. 17.2

Теперь вы видите свою сферу освещенной. Это самый обычный, ненаправленный свет. Далее следует настроить интенсивность света и расстояние, чтобы наша сфера хорошо освещалась.

```
// создаем источник света
pointLight = new THREE.PointLight(0xF8D898);

// позиционируем
pointLight.position.x = -1000;
pointLight.position.y = 0;
pointLight.position.z = 1000;
pointLight.intensity = 2.9;
pointLight.distance = 10000;

// добавляем на сцену
scene.add(pointLight);
```

Рисуем плоскость для игры

Для игровой плоскости создадим меш с типом Plane (Плоскость). Размер плоскости будет соответствовать размеру игровой области с небольшими отступами. Там мы разместим две ракетки - дощечки, которые будут отбивать шарик (рис. 17.3).

```
// создаем материал плоскости
var planeMaterial =
new THREE.MeshLambertMaterial(
{
color: 0x4BD121
});

// создаем игровое поле
var plane = new THREE.Mesh(
new THREE.PlaneGeometry(
```

```
// 95% ширины стола, т.к. нужно показать
// где шар будет выходить за пределы поля
planeWidth * 0.95,
planeHeight,
planeQuality,
planeQuality),
planeMaterial);

scene.add(plane);
```

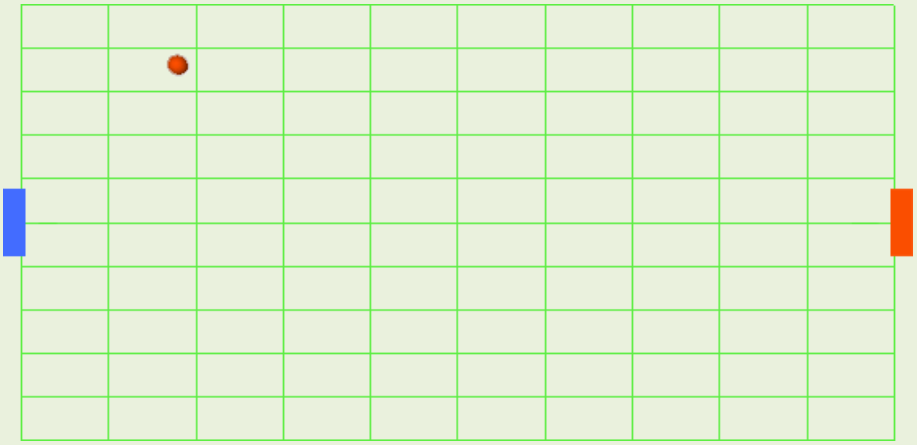

Рис. 17.3

Рисуем дощечки

Дощечки будут мешами с типом Cube (Куб) и расположены напротив друг друга (рис. 17.4).

```
// устанавливаем переменные дощечек
paddleWidth = 10;
paddleHeight = 30;
paddleDepth = 10;
paddleQuality = 1;

// установка дощечки № 1
paddle1 = new THREE.Mesh(
new THREE.CubeGeometry(
paddleWidth,
paddleHeight,
paddleDepth,
paddleQuality,
paddleQuality,
paddleQuality),
paddle1Material);

// добавляем дощечку на сцену
scene.add(paddle1);

// установка дощечки № 2
paddle2 = new THREE.Mesh(
new THREE.CubeGeometry(
paddleWidth,
```

```
paddleHeight,
paddleDepth,
paddleQuality,
paddleQuality,
paddleQuality),
paddle2Material);

// добавляем дощечку на сцену
scene.add(paddle2);

// располагаем дощечки на краю поля напротив друг от друга
paddle1.position.x = -fieldWidth/2 + paddleWidth;
paddle2.position.x = fieldWidth/2 - paddleWidth;

// поднимаем их над игровой поверхностью
paddle1.position.z = paddleDepth;
paddle2.position.z = paddleDepth;
```

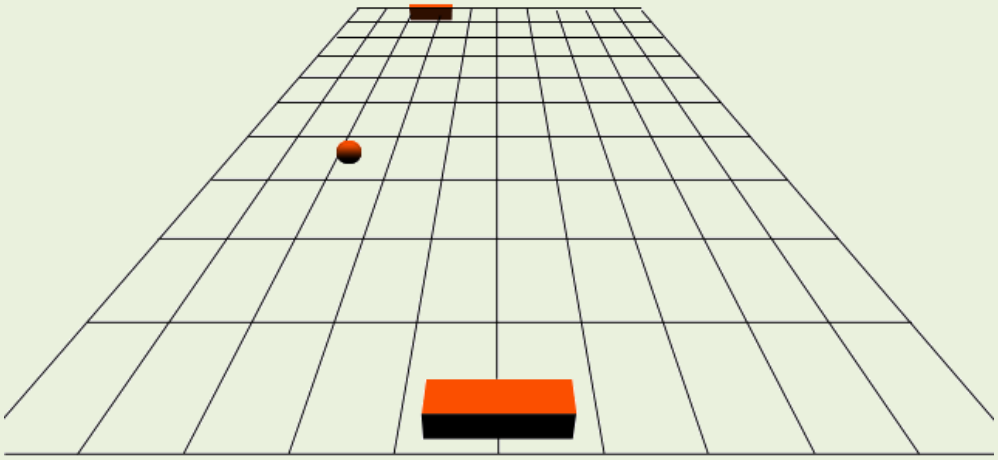

Рис. 17.4

Изменяя значения позиции камеры, можно добиться эффекта перспективы, как показано на рис. 17.4.

17.4. Базовая логика

Движение шарика

Шарик имеет два направления движения - по оси X и по оси Y.

```
// переменные, обозначающие
// направления по осям X и Y и скорость шара
var ballDirX = 1, ballDirY = 1, ballSpeed = 2;
```

По плоскости X в каждом кадре шар будет двигаться с постоянной скоростью, поэтому зададим переменную ballSpeed, которая будет выступать в роли множителя для значений направлений.

```
// обновляем положение шара во время игры
ball.position.x += ballDirX * ballSpeed;
ball.position.y += ballDirY * ballSpeed;
```

Чтобы игра была менее предсказуемой, можно добавить дополнительные характеристики нашему шарику (например, изменять силу удара о дощечку или стену). В частности, мы разрешим шарику двигаться по оси Y со скоростью ballSpeed * 2. Лучше экспериментально настроить это значение для получения приемливого для нас вариант. С помощью этого параметра мы также можем менять сложность игры.

```
// ограничиваем скорость шарика
// чтобы он не летал как сумасшедший
if (ballDirY > ballSpeed * 2)
{
ballDirY = ballSpeed * 2;
}
else if (ballDirY < -ballSpeed * 2)
{
ballDirY = -ballSpeed * 2;
}
```

Логика отскока шара от стен

Во время игры должна происходить проверка: коснулся ли шарик какой-нибудь из боковых стенок. Используя несколько комбинаций if-else, мы проверяем позицию шарика относительно позиций стенок. При столкновении меняем направление шарика по оси Y и тем самым создаем эффект отскакивания шара.

```
// Если шар движется сверху
if (ball.position.y <= -fieldHeight/2)
{
ballDirY = -ballDirY;
}

// Если шар движется снизу
if (ball.position.y >= fieldHeight/2)
{
ballDirY = -ballDirY;
}
```

Позже мы вернемся к этому коду, чтобы реализовать увеличение счета при прохождении шара мимо дощечки.

Управление дощечками при помощи клавиатуры

Из библиотеки keyboard.js, которую мы подключили к нашему главному файлу нам потребуется вызвать только одну функцию – Key.isDown(), которая следит за нажатием клавиш. Получая параметр, фукнция проверяет нажата ли указанная клавиша и возвращает логическое значение (1 или 0).

```
// движение влево
if (Key.isDown(Key.A))
{
// код, двигающий дощечку влево
}
```

Для движения дощечки влево и вправо, будем использовать клавиши A и D соответственно. Если возникнет необходимость хотите использовать другие клавиши, надо открыть и отредактировать файл keyboard.js.

```
...

var Key = {
_pressed: {},

A: 65,
W: 87,
D: 68,
S: 83,
// добавьте код (ASCII) вашей клавиши
// вместе с ее значением, например:

SPACE: 32,

...
};
```

Мы должны предусмотреть, чтобы наша дощечка не вышла за край игровой площадки. Это можно сделать с помощью нескольких комбинаций if-else.

```
// движение влево
if (Key.isDown(Key.A))
{
// двигаем дощечку пока она не коснется стенки
if (paddle1.position.y < fieldHeight * 0.45)
{
paddle1DirY = paddleSpeed * 0.5;
}
// в противном случае мы прекращаем движение и растягиваем
// дощечку чтобы показать, что дальше двигаться нельзя
else
{
paddle1DirY = 0;
paddle1.scale.z += (10 - paddle1.scale.z) * 0.2;
}
}
```

Обратим внимание, что мы используем переменную, в которой хранится направление движения дощечки, вместо того, чтобы просто изменять значение позиции. Это пригодится нам, когда мы будем программировать движение шарика при попадании под углом в движущуюся дощечку.

Программирование игрового робота

При программировании игр приходится создавать игровых противников, противостоящих пользователю игры. В нашей игре мы создадим игорового робота, который просто будет следить за шариком, подставляя дощечку.

```
// применяем функцию Lerp к шару на плоскости Y
paddle2DirY = (ball.position.y - paddle2.position.y) * difficulty;
```

Мы будем менять скорость реакции противника, используя одну переменную. Эта переменная влияет на скорость реакции робота за счет увеличения времени линейной интерполяции (Lerp). При использовании этой функции мы должны сделать так, чтобы робот не имел возможность всегда выигрывать, ограничив его максимальную скорость перемещения. Сделаем это с помощью нескольких комбинаций if-else.

```
// если функция Lerp вернет значение, которое
// больше скорости движения дощечки, мы ограничим его
if (Math.abs(paddle2DirY) <= paddleSpeed)
{
   paddle2.position.y += paddle2DirY;
}
// если значение функции Lerp слишком большое,
// мы ограничиваем скорость paddleSpeed
else
{
 // если дощечка движется в положительном направлении
 if (paddle2DirY > paddleSpeed)
 {
        paddle2.position.y += paddleSpeed;
 }
 // если дощечка движется в отрицательном направлении
 else if (paddle2DirY < -paddleSpeed)
 {
        paddle2.position.y -= paddleSpeed;
 }
}
```

Для добавления оживлюящего сцену визуального эффекта, можно использовать свойство paddle.scale(), которое позволит растянуть дощечку в момент её взаимодействия с шариком или краем стола. При этом надо обеспечить возврат геометрии дощечки к нормальному размеру.

```
// Мы возвращаем значение функции Lerp обратно в 1
// Это используется после растяжения дощечки, которое происходит,
// когда дощечка прикасается к стенкам стола или ударяется о шарик.
// Так мы гарантируем, что она всегда вернется к своему исходному размеру

paddle2.scale.y += (1 - paddle2.scale.y) * 0.2;
```

17.5. Добавляем игровой процесс

Возвращаем шарик после его падения со стола

Текущий вид игрового поля показан на рис. 175.

Рис. 17.5

Чтобы обеспечить главный игровой процесс – ведение счета, нам необходимо убрать эффект отскакивания от двух стенок, где расположены дощечки. Для этого мы просто уберем часть существующего кода.

```
// Если шар двигается сверху
if (ball.position.y <= -fieldHeight/2)
{
   ballDirY = -ballDirY;
}

// Если шар двигается снизу
if (ball.position.y >= fieldHeight/2)
{
   ballDirY = -ballDirY;
}

// ---------------------------------------- //
//         ИЗМЕНЕННЫЙ КОД                   //
// ---------------------------------------- //

// если шар двигается слева (со стороны игрока)
if (ball.position.x <= -fieldWidth/2)
{
   // Компьютер увеличивает счёт
   // Обновляем таблицу с результатами
   // Ставим новый шарик
}

// если шар двигается справа (со стороны компьютера)
if (ball.position.x >= fieldWidth/2)
{
```

```
    // Игрок увеличивает счёт
    // Обновляем таблицу с результатами
    // Ставим новый шарик
}
```

Мы можем реализовать таблицу с результатами разными способами, но для данной игры мы просто увеличим соответствующую переменную.

```
// если шар двигается слева (со стороны игрока)
if (ball.position.x <= -fieldWidth/2)
{
 // Компьютер увеличивает счёт
 score2++;

 // обновляем таблицу с результатами
 document.getElementById("scores").innerHTML = score1 + "-" + score2;

 // устанавливаем новый шар в центр стола
 resetBall(2);

 // проверяем, закончился ли матч (набрано требуемое количество очков)
 matchScoreCheck();
}
```

Мы обновляем содержимое блока со счетом с помощью его свойства innerHTML. После того, как шарик упал, мы должны поставить его снова в центре стола. Напишем простую функцию, которая принимает в качестве параметра дощечку, которая потеряла шарик.

```
// располагаем шарик по центру стола
// также устанавливает скорость и направление шара
// в зависимости от последней победившей стороны

function resetBall(loser)
{
  // располагаем шар по центру стола
 ball.position.x = 0;
 ball.position.y = 0;

 // если игрок проиграл, отправляем шар компьютеру
 if (loser == 1)
 {
        ballDirX = -1;
 }
 // если компьютер проиграл, отправляем шар игроку
 else
 {
        ballDirX = 1;
 }

 // шар двигается в положительном направлении по оси Y (налево от камеры)
 ballDirY = 1;
}
```

Текущий вид игрового поля показан на рис. 17.6.

Рис. 17.6

Отскакивания шара от дощечки

Настало время научить дощечки ударять по шарику. В такой простой игре вся физика шара является всего лишь парой конструкций if-else. Мы проверяем позицию шарика по осям X и Y и сравниваем координаты шарика с границами дощечки. Если они пересекаются, то создаем эффект отскакивания.

```
// если шар имеет одинаковые координаты с дощечкой № 1
// на плоскости X запоминаем позицию ЦЕНТРА объекта
// мы делаем проверку только между передней и средней
// частями дощечки (столкновение одностороннее)

if (ball.position.x <= paddle1.position.x + paddleWidth
&&  ball.position.x >= paddle1.position.x)
{
 // если у шара одинаковые координаты с дощечкой № 1 на плоскости Y

 if (ball.position.y <= paddle1.position.y + paddleHeight/2
 &&  ball.position.y >= paddle1.position.y - paddleHeight/2)
 {
      // шар пересекается с передней частью дощечки
 }
}
```

Также важно проверить направление движение шара, так как мы хотим обеспечить правильный отскок по направлению к противнику.

```
// если шар движется к игроку (отрицательное направление)
if (ballDirX < 0)
{
   // растягиваем дощечку, чтобы показать столкновение
   paddle1.scale.y = 15;

   // меняем направление движения чтобы создать эффект отскакивания шара
   ballDirX = -ballDirX;

   // Меняем угол шара при ударе.
   // Немного усложним игру, позволив скользить шарику
   ballDirY -= paddle1DirY * 0.7;
}
```

Мы также можем влиять на боковое движение шара, основываясь на относительной скорости дощечки в момент удара по ней. Это особенно полезно при учете дополнительных факторов, таких как скольжение. Скольжение шара часто является единственным способом переиграть противника, поэтому оно важно в этой игре.

Не забудем продублировать код, обновив значения в соответствии с положением дощечки противника. Окончательный вариант функции, описывающей столкновение дощечки с шариком:

```
// Логика столкновений с дощечкой
function paddlePhysics()
{

 // ЛОГИКА ДЛЯ ДОЩЕЧКИ ИГРОКА

 // если шар имеет одинаковые координаты с дощечкой № 1 на плоскости X
 // запоминаем позицию ЦЕНТРА объекта, при этом мы проверяем
 // только переднюю и среднюю части дощечки (одностороннее столкновение)

 if (ball.position.x <= paddle1.position.x + paddleWidth
 &&  ball.position.x >= paddle1.position.x)
 {
          // и если шар имеет одинаковые координаты с дощечкой № 1 на плоскости Y
          if (ball.position.y <= paddle1.position.y + paddleHeight/2
          &&  ball.position.y >= paddle1.position.y - paddleHeight/2)
          {
            // и если шар направляется к игроку (отрицательное направление)
            if (ballDirX < 0)
            {
                    // растягиваем дощечку при ударе
                    paddle1.scale.y = 15;
                    // меняем направление движения шара (эффект отскакивания)
                    ballDirX = -ballDirX;
                    // меняем угол шара при ударе
                    ballDirY -= paddle1DirY * 0.7;
            }
          }
 }
```

```
// ЛОГИКА ДЛЯ ДОЩЕЧКИ СОПЕРНИКА

// если шар имеет одинаковые координаты с дощечкой # 2 на плоскости X
// запоминаем позицию ЦЕНТРА объекта, при этом мы проверяем
// только переднюю и среднюю части дощечки (одностороннее столкновение)
if (ball.position.x <= paddle2.position.x + paddleWidth
&&  ball.position.x >= paddle2.position.x)
{
        // и если шар имеет одинаковые координаты с дощечкой № 2 на плоскости Y
        if (ball.position.y <= paddle2.position.y + paddleHeight/2
        &&  ball.position.y >= paddle2.position.y - paddleHeight/2)
        {
          // и если шар направляется к сопернику (положительное направление)
          if (ballDirX > 0)
          {
                    // растягиваем дощечку при ударе
                    paddle2.scale.y = 15;
                    // меняем направление движения шара (эффект отскакивания)
                    ballDirX = -ballDirX;
                    // меняем угол шара при ударе
                    ballDirY -= paddle2DirY * 0.7;
          }
        }
}
}
```

Ведение счёта

В пинг-понге для победы нужно набрать заданное количество очков. Создадим для этого переменную maxScore.

```
// переменные с очками каждого игрока
var score1 = 0, score2 = 0;

// игра завершится, когда кто-то наберет 7 очков
var maxScore = 7;
```

Создадим функцию matchScoreCheck(), которая будет вызываться когда кто-то не отбил шарик и проверять очки каждого игрока.

```
// проверяем, закончился ли матч (набрано требуемое количество очков)
function matchScoreCheck()
{
   // если выиграл игрок
   if (score1 >= maxScore)
   {
      // останавливаем шар
      ballSpeed = 0;
      // выводим текст
      document.getElementById("scores").innerHTML = "Победа игрока!";
      document.getElementById("winnerBoard").innerHTML = "Обновите страницу
          чтобы сыграть снова";
   }
```

```
    // если выиграл компьютер
    else if (score2 >= maxScore)
    {
        // останавливаем шар
        ballSpeed = 0;
        // выводим текст
        document.getElementById("scores").innerHTML = "Компьютер выиграл!";
        document.getElementById("winnerBoard").innerHTML = "Обновите страницу
            чтобы сыграть снова";
    }
}
```

После того, как игра закончена, нужно поставить шарик в центр стола и прекратить движение.

17.6. Дорабатываем игру

Интерфейс

Чтобы игрок был в курсе того, что происходит, необходимо отображать для него игровую информацию. Мы будем показывать счет игры на экране (рис. 17.7). Чтобы не отрисовывать интерфейс в том же окне, что и игра, воспользуемся специально созданными для этого тегами.

Рис. 17.7

Покажем сколько очков нужно набрать для победы – за это у нас отвечает блок с id=winnerBoard, который мы будем обновлять при запуске игры.

```
// Обновляем блок, содержащий сообщение о необходимых для победы очках
document.getElementById("winnerBoard").innerHTML = "Набравший " + maxScore + "
        очков победит!";
```

Тени

Для придания реалистичности сцене используем световые эффекты (рис. 17.8). Библиотека Three.js обладает разными возможности для создания теней для примитивов – кубов, плоскостей, сфер и т.д. Тени не могут быть созданы только при точечном освещении, поэтому нужно добавить источник направленного света и прожектор. Напомним: прожектор светит большим круглым лучом света на поверхность, в то время как направленный источник просто излучает свет в определенном направлении без учета позиции относительно объекта.

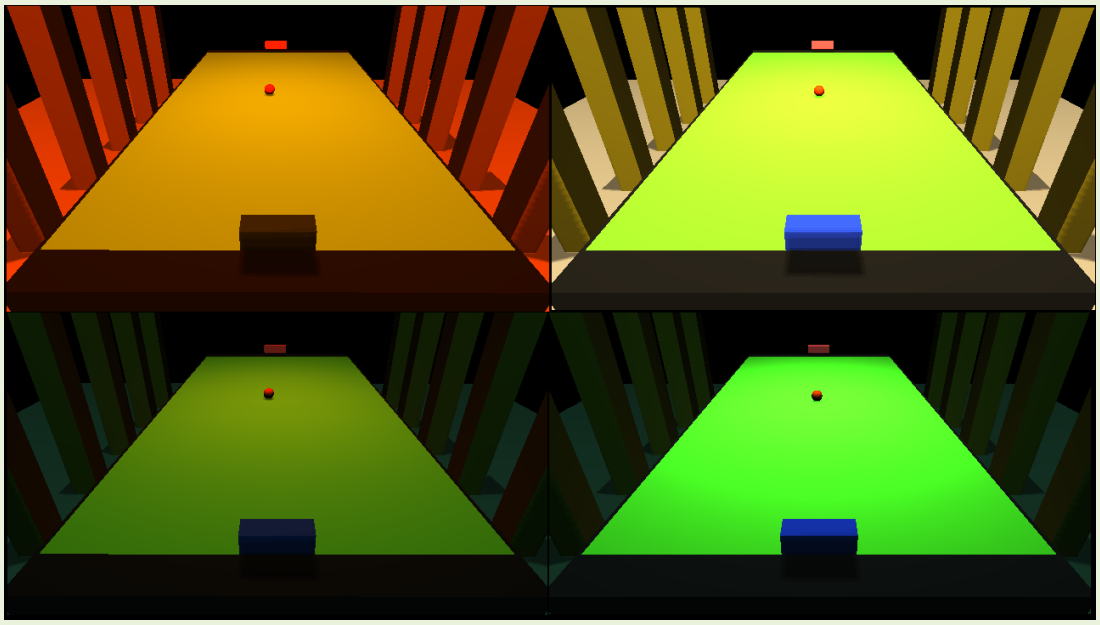

Рис. 17.8

Мы будем использовать прожектор, потому что он явно указывает откуда падает свет и в какую сторону направлен.

```
// создаем точечный свет
pointLight = new THREE.PointLight(0xF8D898);

// позиционируем
pointLight.position.x = -1000;
pointLight.position.y = 0;
pointLight.position.z = 1000;
pointLight.intensity = 2.9;
pointLight.distance = 10000;
scene.add(pointLight);

// добавляем прожектор для создания теней
spotLight = new THREE.SpotLight(0xF8D898);
spotLight.position.set(0, 0, 460);
spotLight.intensity = 1.5;
spotLight.castShadow = true;
scene.add(spotLight);

// включаем рендеринг теней
renderer.shadowMapEnabled = true;
```

Чтобы придать динамичности нашей игре, сделаем чтобы луч прожектора двигался за катящимся шаром.

```
// позиционируем тени
// также как и шарик
spotLight.position.x = ball.position.x;
spotLight.position.y = ball.position.y;
```

Чтобы объекты отбрасывали тени, или тени появлялись на объектах, установим значение true для переменных .receiveShadow и .castShadow соответствующих объектов (рис. 17.9):

```
paddle1 = new THREE.Mesh(
    new THREE.CubeGeometry(paddleWidth, paddleHeight,
        paddleDepth, paddleQuality,
        paddleQuality, paddleQuality),
    paddle1Material);

// добавляем сферы на сцену
scene.add(paddle1);

paddle1.receiveShadow = true;
paddle1.castShadow = true;
```

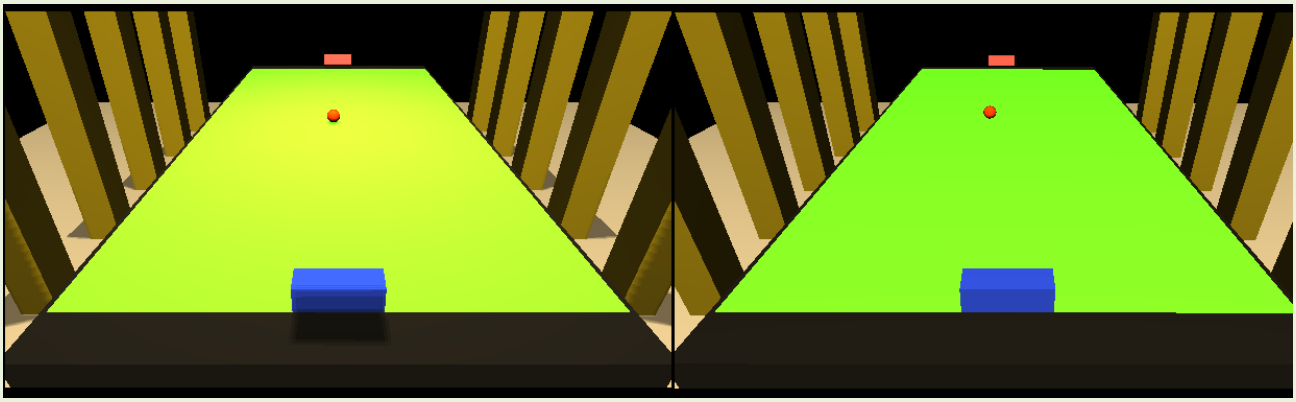

Рис. 17.9

Полный код игры можно посмотреть в файлах **ex17_01.html**, **ex17_01.js**.

Итоги

При разработке нашей простой игры с испрользованием библиотеки Three.js мы создали логику игры пинг-понг, реализовали перемещение и взаимодействие объектов, применили световые эффекты. Несколько улучшить визуальные свойства игры можно, например, усовершенствовав индикатор для отображения игровой информации и переместив его на сцену, чтобы позволить играть в полноэкранном режиме. Можно также поработать со сложными шейдерами, чтобы создать отражения и другие визуальные эффекты.

Контрольные вопросы

1. Структура программы.
2. Последовательность формирования 3D-сцены.
3. Добавление объектов к сцене.
4. Добавление управления сценой.
4. Добавление визуальных эффектов.