Методическое пособие по выполнению модуля «Игра»

Автор: Фунтиков Михаил Николаевич

2020 год.

Содержание

Общие требования

Тема: «Выполнение конкурсного задания по модулю «Игра»

Цель: получение навыков создания браузерной игры с применением языка программирования JavaScript.

Инструкция:

1) Изучите предложенное конкурсное задание;
2) Следуя инструкциям, настройте свое рабочее место;
3) Разработайте логику перехода страниц;
4) Разработайте логику главной страницы;
5) Разработайте логику поведения игрока;
6) Разработайте логику поведения фруктов и коллизию с игроком;
7) Разработайте логику учёта очков и отнимания жизней;
8) Разработайте логику поведения таймера;
9) Разработайте логику последнего экрана;
10) Разработайте логику применения паузы;
11) Разработайте логику автоматического сбора фруктов.

Конкурсное задание

Введение

К вам обратилась компания по разработке игр для веб-сайтов. Компания просит помочь в разработке веб-игры. Вам предоставляется вся необходимая верстка. Ваша задача – только клиентское программирование.

Время на выполнение: 1,5ч.

Описание проекта и задач

Экран входа

При открытии игры должен быть отображен экран входа в игру. На данном экране присутствует поле для указания вашего имени и кнопка входа.

Если имя пользователя не указано, то кнопка должна быть не активна. Когда имя пользователя будет заполнено, кнопка должна стать активной.

В поле для имени игрока должно быть указано имя последнего игрока, который играл в игру (даже после перезагрузки страницы). Если никто еще не играл в игру, то поле должно быть пустым.

При клике на кнопку входа экран должен смениться на игровой.

Игровой экран

На игровом экране находиться имя пользователя, которое должно корректно отображаться, секундомер (начинает считать с 00:00), который должен быть запущен при старте игры, счетчик жизней (при старте у игрока есть 3 жизни) и счетчик пойманных предметов.

На игровом поле есть корзина, в которую нужно ловить падающие фрукты.

При старте игры корзина должна находиться снизу по центру.

В процессе игры должны появляться фрукты (1 фрукт каждую секунду).

Фрукты должны падать вниз с разной скоростью.

Игрок должен иметь возможность управлять корзиной с помощью стрелок влево и вправо.

Корзина не должна выходить за пределы игровой зоны.

Если фрукт достигает земли (нижняя граница экрана), то у пользователя отнимаются жизни.

Если удерживать «пробел», то все фрукты, присутствующие на экране будут считаться собранными. Этот процесс может быть активен не дольше 4 секунд. После этого нужно 5 секунд на восстановление.

Постарайтесь отобразить процесс «автоматического сбора».

Должна быть возможность поставить игру на паузу нажав кнопку ESC. Повторное нажатие должно продолжить игру. Во время паузы все интерактивные действия (анимация, секундомер, фрукты, корзина) должны быть приостановлены.

Когда жизни будут равны нулю, то игра должна закончиться и должен отобразиться экран с результатами.

Экран с результатами

На экране с результатами необходимо отобразить время, которое игрок продержался в игре и кол-во собранных фруктов. Если игрок продержался дольше 10 секунд, то должно отобразиться сообщение о выигрыше, иначе о проигрыше.

На экране с результатами есть кнопка «Играть сначала» при клике на которую игра должна перезапуститься.

Инструкции для участника

Ваша работа должна быть доступна по адресу: http://xxxxxx-m2.wsr.ru, где xxxxxx – ваш логин

Вы можете использовать библиотеку jQuery.

Раздел 1. Настройка своего рабочего места

Для работы нам понадобиться браузер Google Chrome и любой текстовый редактор (Notepad++, Sublime). Можно использовать полноценные IDE: Atom, WebStorm, PhpStorm.

Скачиваем архив layout.rar и распаковываем. Затем открываем весь каталог в текстовом редакторе или IDE (пример смотрите на рисунке 1.1). Файл index.html открываем в браузере (рисунок 1.2).

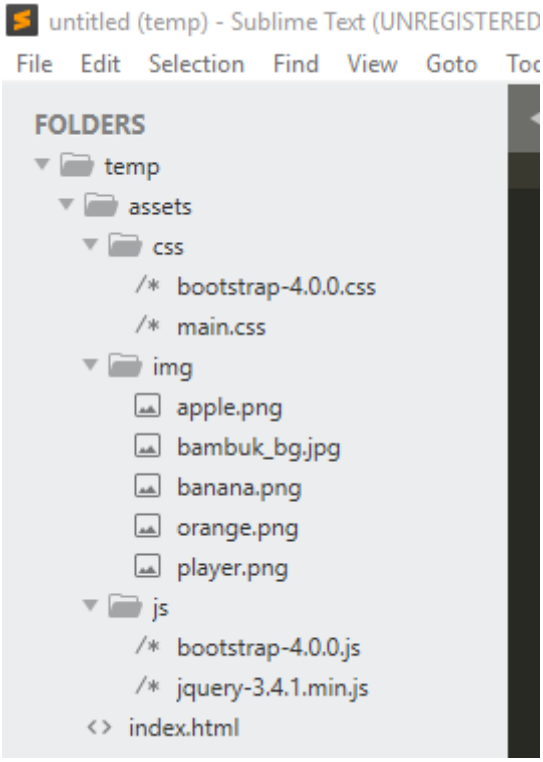

Рисунок 1.1 – Пример открытия проекта в Sublime

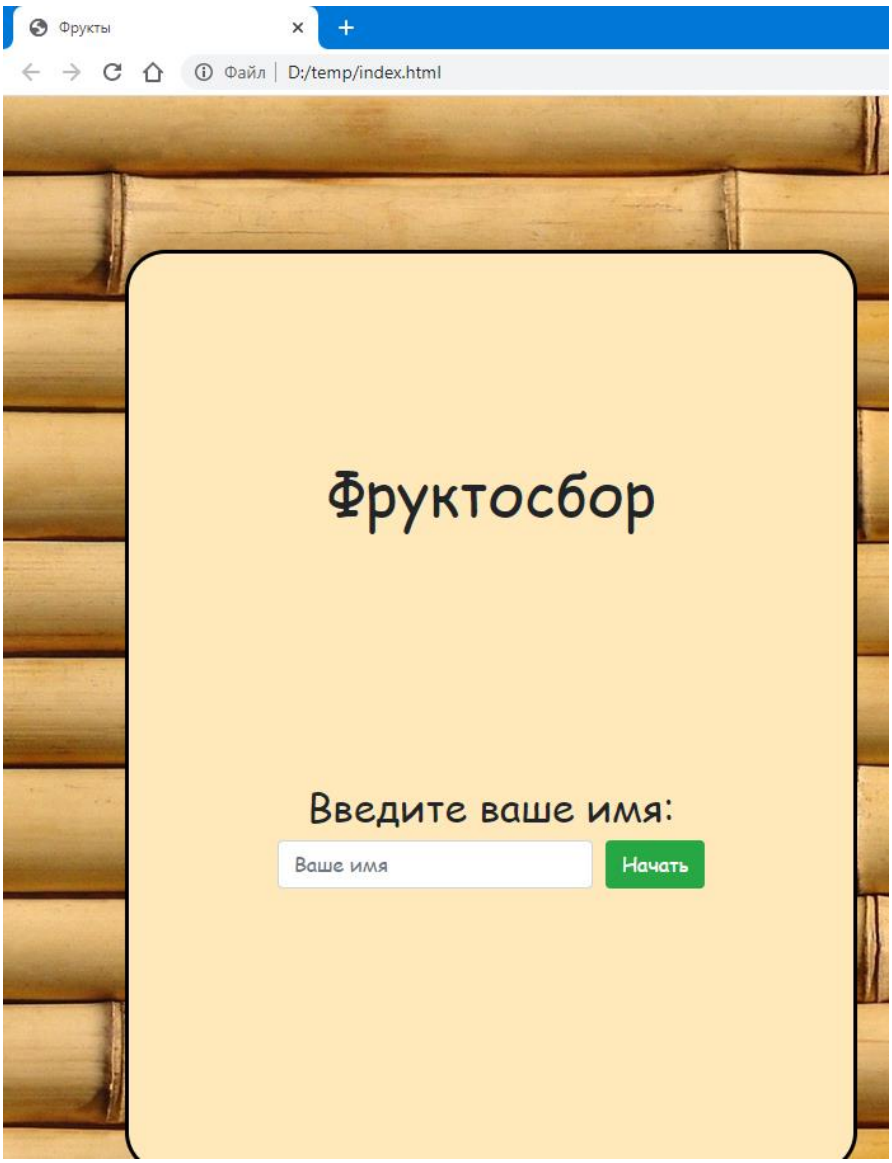

Рисунок 1.2 – Пример открытого проекта в браузере Google Chrome

Создадим 2 файла в каталоге assets/js: main.js и game.js. В них мы далее будем реализовывать логику (рисунок 1.3). Теперь подключим наши файлы в index.html (смотрите рисунок 1.4).

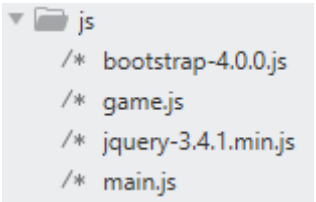

Рисунок 1.3 – Созданные файлы в каталоге js

Рисунок 1.4 – Подключение файлов в index.html

Теперь мы готовы для реализации логики игры.

Раздел 2. Разработка логики перехода страниц

Объявим класс Game с пустым конструктором и единственным методом start в файле game.js.

```
class Game {
        constructor() {

        }
        start () {

        }
}
```

Теперь в файле main.js объявим две переменные.

```
let game = {
    game: []
}
let panel = 'start';
```

Затем в этом же файле распишем функцию навигации, которая срабатывает по клику мышки на кнопках. Расписываем мы её сразу с рестартом игры, чтобы потом не возвращаться к ней.

```
let nav = () => {
    document.onclick = (e) => {
        e.preventDefault();
        switch (e.path[0].id) {
            case "startGame":
                go('game', 'd-block');
                break;
            case "restart":
                go('game', 'd-block');
                $('.elements').remove();
                $("#game").append(`<div class="elements"></div>`);
                break;
        }
    }
}
```

Нужно написать вызываемую функцию go, которая будет отслеживать на какой странице мы находимся в данный момент. Обратите внимание, что $(`#${page}`) пишется с обратными кавычками (левая клавиша от 1). И в будущем внимательно смотрите не такие конструкции, так как таким образом мы будем избегать стандартной конкатенации через знак «+».

```
let go = (page, attribute) => {

    let pages = ['start', 'game', 'end'];

    panel = page;

    $(`#${page}`).attr('class', attribute);

    pages.forEach(e => {
        if (page != e) {
            $(`#${e}`).attr('class', 'd-none');
        }
    })
}
```

Запишем функцию отслеживания страниц в интервале.

```
let startLoop = () => {
    let inter = setInterval(()=>{
        if(panel !== "start"){
            clearInterval(inter);
        }
    },100);
}
```

Теперь в файле game.js добавим запуск наших функций при загрузке окна браузера.

```
window.onload = () => {
    nav();
    startLoop();
    setInterval(() => {
        if (panel === "game") {
            game.game = new Game();
            game.game.start();
            panel = "game process";
        }
    }, 500)
};
```

На этом создание переходов по страницам заканчивается. Если попробовать нажать кнопку «Начать» на начальном экране, то должен осуществляться переход на игровое поле. Если этого не происходит, то откройте консоль разработчика F12 и в разделе консоль вам покажет где у вас ошибка (смотрите пример на рисунке 2.1). Смотрите на красный шрифт и справа будет указана строка, где допущена ошибка.

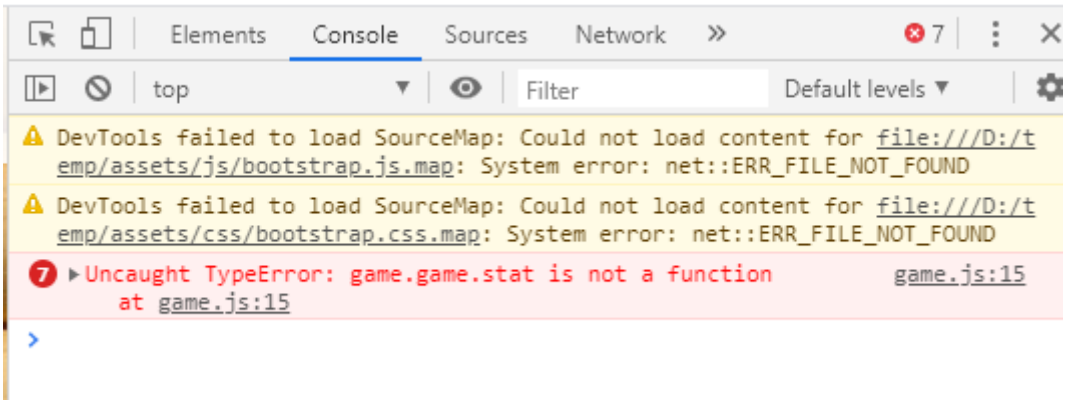

Рисунок 2.1 – Пример отслеживания ошибки через консоль разработчика.

Раздел 3. Разработка логики главной страницы

Реализация логики в этом разделе будет последний раз обращаться к файлу main.js, в последующих разделах будем работать только с логикой игры, а значит с файлом game.js.

Добавим функцию проверки localstorage на наличие уже введенного имени игрока. В файл game.js допишем её вызов.

```
window.onload = () => {
    checkStorage();
    nav();
    startLoop();
```

Вернёмся в файл main.js и объявим ещё одну переменную, в которой будем хранить имя игрока.

```
let name = '';
let game = {
    game: []
}
let panel = 'start';
```

Теперь напишем функцию проверки localstorage, которая в случае не первого входа пользователя будет автоматически заполнять поле ввода для имени игрока.

```
let checkStorage = () => {
    if(localStorage.getItem('userName') != null) {
        $(`#nameInput`).val(localStorage.getItem('userName'));
    }
}
```

Напишем функцию, которая будет проверять, ввел ли пользователь имя игрока и блокировать кнопку «Начать», если поле пустое или стоят просто пробелы.

```
let checkName =  () => {
    name = $(`#nameInput`).val().trim();
    if(name != ""){
        localStorage.setItem('userName', name);
        $(`#startGame`).attr('disabled',false);
    }
    else{
        $(`#startGame`).attr('disabled',true);
    }
}
```

Осталось вызывать функцию checkName в startLoop. Это сделано для того, чтобы если пользователь ввёл имя, а затем стёр его, то кнопка «Начать» становилась снова не активной.

```
let startLoop = () => {
    let inter = setInterval(()=>{
        if(panel !== "start"){
            clearInterval(inter);
        }
        checkName();
    },100);
}
```

Теперь можете проверить работоспособность алгоритма. Если вы введёте имя, а затем обновите страницу браузера (F5), то имя игрока будет вставляться автоматически.

Больше мы не будем возвращаться к файлу main.js. В последующих разделах будем, если мы говорим о JavaScript, то пишем его только в файле game.js.

Совет: обновлять окно браузера рекомендуется при разработке на сочетание клавиш Ctrl+F5 или Ctrl+R. Чтобы обновлялся кэш файлов.

Раздел 4. Разработка логики поведения игрока

Для начала давайте немного доделаем логику прошлого раздела, а именно будем отображать имя игрока на игровом экране. Для этого объявим один параметр в классе Game и в методе start вызовем метод loop.

```
class Game {
        constructor() {
            this.name = name;
        }
        start () {
            this.loop();
        }
```

Напишем метод loop, который будет работать через requestAnimationFrame, для создания плавной анимации. А внутри метода создадим рекурсию. И вызовем метод установки параметров setParams.

```
    loop() {
    requestAnimationFrame(() => {
            this.setParams();
            this.loop();
    })
}
```

Осталось написать метод setParams, который не данный момент будет вставлять имя на игровое поле (в дальнейшем мы её будем дополнять).

```
setParams() {
    let params = ['name'];
    let value = [this.name];

    params.forEach((e, i) => {
        $(`#${e}`).html(value[i]);
    })
}
```

Можно переключиться на браузер и посмотреть, что имя теперь передается на игровое поле.

Напишем класс который у нас будет отвечать за отображение, назовём его Drawable. В нём напишем конструктор, входящим параметром в который будет экземпляр класса Game. Зададим положения по осям x и y равные 0. Ширину и высоту тоже зададим равными 0. И «скорость перемещения» по x и y тоже обнулим.

```
class Drawable {
    constructor(game) {
        this.game = game;
        this.x = 0;
        this.y = 0;
        this.h = 0;
        this.w = 0;
        this.offsets = {
            x: 0,
            y: 0
        }
    }
}
```

Вернёмся к классу Game и зададим ещё один параметр, который позволит нам отслеживать игровую зону.

```
class Game {
        constructor() {
                this.name = name;
            ——>this.$zone = $('.elements');
        }
```

В классе Drawable создадим метод createElement, который нам позволит создавать игровые элементы на игровом поле.

```
createElement() {
    this.$element = $(`<div class="element ${this.constructor.name.toLowerCase()}"></div>`);
    this.game.$zone.append(this.$element);
}
```

Метод update будет перемещать наши объекты в зависимости от параметров «скорости». Его мы тоже пишем в класс Drawable.

```
update() {
    this.x += this.offsets.x;
    this.y += this.offsets.y;
}
```

Последний метод из класс Drawable поможет нам отображать перемещение объекта пользователю, за счёт изменения CSS свойств.

```
draw() {
    this.$element.css({
        left: this.x + "px",
        top: this.y + "px",
        width: this.w + "px",
        height: this.h + "px"
    })
}
```

Нужно создать в классе Game ещё один параметр, в котором мы будем хранить массив объектов на игровом поле.

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
        ——> this.elements = [];
        }
```

Создадим метод updateElements в классе Game, который будет вызывать методы update и draw для каждого объекта.

```
updateElements() {
    this.elements.forEach(e => {
        e.update();
        e.draw();
    })
}
```

Нужно постоянно вызывать этот метод, а поэтому мы его будем вызывать через метод loop

```
loop() {
requestAnimationFrame(() => {
——> this.updateElements();
        this.setParams();
        this.loop();
```

Теперь создадим класс Player, который будет дочерним классом Drawable. В конструктор мы будем принимать экземпляр класса Game. Затем наследуем все параметры родительского класса и назначаем свои параметры размеров. Положение мы определяем исходя из параметров игрового поля. В конце мы вызовем родительский метод createElement

```
class Player extends Drawable {
    constructor(game) {
        super(game);
        this.w = 244;
        this.h = 109;
        this.x = this.game.$zone.width() / 2 - this.w / 2;
        this.y = this.game.$zone.height() - this.h;
        this.createElement();
    }
}
```

Сейчас мы посмотрим в браузере, то у нас не будет не ошибок, но и корзину мы не увидим. Поэтому в классе Game напишем генерацию объектов через метод generate.

```
generate(className) {
        let element = new className(this);
        this.elements.push(element);
        return element;
}
```

И вызовем метод generate задав новый параметр, где мы будем хранить экземпляр класс Player.

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
                this.elements = [];
        ——> this.player = this.generate(Player);
        }
```

Можете проверить браузер. Вы должны увидеть корзину, но двигать мы её пока не можем. Для этого зададим ещё несколько параметров для Player. Первый будет отвечать за скорость. Второй позволит нам отслеживать нажатие клавиш. Вызовем в конструкторе ещё один метод bindKeyEvents.

```
class Player extends Drawable {
    constructor(game) {
        super(game);
        this.w = 244;
        this.h = 109;
        this.x = this.game.$zone.width() / 2 - this.w / 2;
        this.y = this.game.$zone.height() - this.h;
        this.speedPerFrame = 20;
        this.keys = {
            ArrowLeft: false,
            ArrowRight: false
        };
        this.createElement();
        this.bindKeyEvents();
    }
```

Нужно написать метод bindKeyEvents. Его задача будет менять статус кнопки (нажата или отпущена) и вызывать метод changeKeyStatus, который будет за это отвечать.

```
bindKeyEvents() {
    document.addEventListener('keydown', ev => this.changeKeyStatus(ev.code, true));
    document.addEventListener('keyup', ev => this.changeKeyStatus(ev.code, false));
}

changeKeyStatus(code, value) {
    if (code in this.keys) {
        this.keys[code] = value;
    }
}
```

Нужно отобразить пользователю движение нашей корзины. Для этого используем метод update, в котором будем отслеживать какая клавиша нажата и двигать корзину в соответствующем направлении. Сразу в этих же условиях через логический оператор напишем ограничение размера поля, чтобы корзина не выезжала за границы. Затем вызовем родительский метод update.

```
update() {
    if (this.keys.ArrowLeft && this.x > 0) {
        this.offsets.x = -this.speedPerFrame;
    } else if (this.keys.ArrowRight && this.x < this.game.$zone.width() - this.w) {
        this.offsets.x = this.speedPerFrame;
    } else {
        this.offsets.x = 0;
    }
    super.update();
}
```

Теперь корзина готова и она должна двигаться согласно условиям задания.

Раздел 5. Разработка логики поведения фруктов и коллизия с игроком

Для решения задачи создадим новый класс Fruits, который будет дочерним классом от Drawable. Зададим начальные параметры в конструкторе, а также напишем метод update.

```
class Fruits extends Drawable {
    constructor(game) {
        super(game);
        this.w = 70;
        this.h = 70;
        this.x = random(0, this.game.$zone.width() - this.w);
        this.y = 60;
        this.offsets.y = 3;
        this.createElement();
    }

    update() {
        super.update();
    }

}
```

Мы должны написать функцию random, которой мы ещё будем пользоваться. Описать её нужно вне классов.

```
let random = (min, max) => {
    min = Math.ceil(min);
    max = Math.floor(max);

    return Math.floor(Math.random() * (max - min + 1)) + min;
};
```

Затем создадим 3 дочерних класса от Fruits, в которых мы будем только менять «скорость падения»: Apple, Banana, Orange.

```
class Apple extends Fruits {
    constructor(game) {
        super(game);
        this.offsets.y = 5;
    }
}

class Banana extends Fruits {
    constructor(game) {
        super(game);
    }
}

class Orange extends Fruits {
    constructor(game) {
        super(game);
        this.offsets.y = 7;
    }
}
```

Вернёмся в класс Game и зададим там, два новых параметра. Первый fruits будет хранить массив названия всех классов, а второй counterForTimer поможет нам с генерацией фруктов раз в секунду (в будущем мы через него сделаем секундомер).

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
                this.elements = [];
                this.player = this.generate(Player);
                this.fruits = [Apple, Banana, Orange];
                this.counterForTimer = 0;
        }
```

Теперь ещё раз модифицируем метод loop, где мы сделаем вызов функции генерации фрукта раз в секунду.

```
loop() {
        requestAnimationFrame(() => {
                this.counterForTimer++;
                if (this.counterForTimer % 60 === 0) {
                        this.randomFruitGenerate();
                }
                this.updateElements();
                this.setParams();
                this.loop();
        })
}
```

Опишем поведение функции randomFriutGenerate (в классе Game), который будет выглядеть так. Выбираем случайный индекс из массива классов фруктов, затем вызываем метод generate, в который предаём имя класса со случайным индексом.

```
randomFruitGenerate() {
    let ranFruit = random(0, 2);
    this.generate(this.fruits[ranFruit]);
}
```

Можете проверить в браузере, что у вас начали появляться случайные фрукты. Было бы не плохо их анимировать, а для этого в файл main.css напишем правило поведения анимации @keyframes, которое будет вращать наши фрукты на 360 градусов (напишем в самом низу файла).

```
@keyframes rotate {
    to {
        transform: rotate(360deg);
    }

}
```

Добавим каждому фрукту свою скорость отработки анимации, для создания ещё большего эффекта разницы скорости их падения.

```
.apple{
    background: url("../img/apple.png") center no-repeat;
    background-size: contain;
    animation: rotate infinite .6s linear;
}
.orange{
    background: url("../img/orange.png") center no-repeat;
    background-size: contain;
    animation: rotate infinite .7s linear;
}
.banana{
    background: url("../img/banana.png") center no-repeat;
    background-size: contain;
    animation: rotate infinite .8s linear;
}
```

Осталось написать коллизию фруктов с корзиной. Для этого в классе Drawable напишем метод isCollision, который будет возвращать нам соприкосновение по диагоналям объектов.

```
isCollision(element) {
    let a = {
        x1: this.x,
        x2: this.x + this.w,
        y1: this.y,
        y2: this.y + this.h,
    };

    let b = {
        x1: element.x,
        x2: element.x + element.w,
        y1: element.y,
        y2: element.y + element.h,
    };
    return a.x1 < b.x2 && b.x1 < a.x2 && a.y1 < b.y2 && b.y1 < a.y2;
}
```

Теперь перепишем метод update у класса Fruits, в котором мы начнём проверять коллизию и если она есть, то выводить надпись в консоль разработчика.

```
update() {
    if (this.isCollision(this.game.player) && this.offsets.y > 0) {
        console.log('есть коллизия!');
    }
    super.update();
}
```

Теперь мы сделали коллизию и можем проверить её через браузер. В следующем разделе мы её реализуем до конца.

Раздел 6. Разработка логики учёта очков и отнимания жизней

Напишем ещё 2 параметра для класса Game, которые будут учитывать наши очки и жизни.

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
                this.elements = [];
                this.player = this.generate(Player);
                this.fruits = [Apple, Banana, Orange];
                this.counterForTimer = 0;
                this.points = 0;
                this.hp = 3;
        }
```

Перепишем метод setParams класса Game, который будет нам позволять отслеживать изменения очков и жизней на игровом поле (больше к этому методу не возвращаемся).

```
setParams() {
    let params = ['name', 'points', 'hp'];
    let value = [this.name, this.points, this.hp];
    params.forEach((e, i) => {
        $(`#${e}`).html(value[i]);
    })
}
```

Необходимо так же удалять элементы с игрового поля после коллизии, поэтому сразу опишем этот метод в классе Game.

```
remove(el) {
        let idx = this.elements.indexOf(el);
        if (idx !== -1) {
                this.elements.splice(idx, 1);
                return true;
        }
        return false;
}
```

Добавим метод removeElement в класс Drawable.

```
removeElement() {
    this.$element.remove();
}
```

Заменим вывод в консоль в методе update класса Fruits вызовом функции takePoint.

```
update() {
    if (this.isCollision(this.game.player) && this.offsets.y > 0) {
        this.takePoint(this.game.element);
    }
    super.update();
}
```

Опишем поведение метода takePoint класса Fruits, который будет прибавлять 1 балл, если фрукт пойман в корзину. Фрукт будет исчезать, так как мы будем вызывать функцию удаления. Если бы мы её не написали, то счётчик быстро прибавлял баллы на всё время коллизии.

```
takePoint() {
    if (this.game.remove(this)) {
        this.removeElement();
        this.game.points++;
    }
}
```

Допишем ещё одно условие для метода update класса Fruits, которое будет учитывать коллизию с низом экрана и вызывать метод takeDamage.

```
update() {
    if (this.isCollision(this.game.player) && this.offsets.y > 0) {
        this.takePoint(this.game.element);
    }
    if (this.y > this.game.$zone.height()) {
        this.takeDamage(this.game.element);
    }
    super.update();
}
```

Метод takeDamage класса Fruits будет отнимать 1 жизнь и удалять фрукт.

```
takeDamage() {
    if (this.game.remove(this)) {
        this.removeElement();
        this.game.hp--;
    }
}
```

Все счётчики работают, пока нам этого достаточно. В браузере видно, что жизни уходят в отрицательное значение, но мы это исправим, когда будем реализовывать последний экран.

Раздел 7. Разработка логики поведения таймера

Для реализации нам нужен объект с параметрами в классе Game, добавим его.

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
                this.elements = [];
                this.player = this.generate(Player);
                this.fruits = [Apple, Banana, Orange];
                this.counterForTimer = 0;
                this.points = 0;
                this.hp = 3;
                this.time = {
                    m1: 0,
                    m2: 0,
                    s1: 0,
                    s2: 0
                };
        }
```

В методе loop напишем вызов метода таймера, который будет работать раз в секунду.

```
loop() {
        requestAnimationFrame(() => {
                this.counterForTimer++;
                if (this.counterForTimer % 60 === 0) {
                    this.timer();
                    this.randomFruitGenerate();
                }
                this.updateElements();
                this.setParams();
```

Напишем метод, который будет нам выводить секундомер.

```
timer() {
        let time = this.time;
        time.s2++;
        if (time.s2 >= 10) {
            time.s2 = 0;
            time.s1++;
        }
        if (time.s1 >= 6) {
            time.s1 = 0;
            time.m2++;
        }
        if (time.m2 >= 10) {
            time.m2 = 0;
            time.m1++;
        }
        let str = `${time.m1}${time.m2}:${time.s1}${time.s2}`;
        $("#timer").html(str);
    }
```

Раздел 8. Разработка логики последнего экрана

Нам нужен ещё один параметр для Game, который поможет останавливать игру, если жизни станут равны нулю.

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
                this.elements = [];
                this.player = this.generate(Player);
                this.fruits = [Apple, Banana, Orange];
                this.counterForTimer = 0;
                this.points = 0;
                this.hp = 3;
                this.time = {
                    m1: 0,
                    m2: 0,
                    s1: 0,
                    s2: 0
                };
        ----->  this.ended = false;
        }
```

Теперь модифицируем метод loop. Он будет вызывать метод end в случае, если жизней станет 0. А также обернём рекурсивный вызов loop в условие, которое следит об окончании игры.

```
loop() {
         requestAnimationFrame(() => {
                 this.counterForTimer++;
                 if (this.counterForTimer % 60 === 0) {
                         this.timer();
                         this.randomFruitGenerate();
                 }
         ----->  if (this.hp <= 0) {
         ----->          this.end();
         ----->  }
         ----->  if (!this.ended) {
         ----->          this.loop()
         ----->  }
                 this.updateElements();
                 this.setParams();
         })
}
```

Напишем метод end. Будем сразу при его вызове сообщать, что игра окончена, изменяя параметр ended. Затем условием проверяем продержался ли пользователь больше 10 секунд и выдавать соответствующие данные на последнем экране.

```
end() {
    this.ended = true;
    let time = this.time;
    if (time.s1 >= 1 || time.m2 >= 1 || time.m1 >= 1) {
        $('#playerName').html(`Поздравляем, ${this.name}!`);
        $('#endTime').html(`Ваше время: ${time.m1}${time.m2}:${time.s1}${time.s2}`);
        $('#collectedFruits').html(`Вы собрали ${this.points} фруктов`);
        $('#congratulation').html(`Вы выиграли!`);
    } else {
        $('#playerName').html(`Жаль, ${this.name}!`);
        $('#endTime').html(`Ваше время: ${time.m1}${time.m2}:${time.s1}${time.s2}`);
        $('#collectedFruits').html(`Вы собрали ${this.points} фруктов`);
        $('#congratulation').html(`Вы проиграли!`);
    }
    go('end', 'panel d-flex justify-content-center align-items-center');
}
```

Обратите внимание, что в условии учтены минуты тоже. Это сделано для того, чтобы при ситуации 01:00 например или 10:00 пользователь не проиграл.

Кнопка рестарт работает, так как необходимо по заданию. Мы её написали уже давно, когда писали функцию nav в файле main.js.

Раздел 9. Разработка логики применения паузы

Нам необходимо добавить блоки паузы в файл index.html. Добавим их сразу после открывающего тэга body до открывающегося тэга main.

```
<body>
<div class="pause">
    <h1>Пауза</h1>
    <p>Нажмите [Esc] для отключения</p>
</div>
<main>
```

В файле main.css напишем свойства.

```
.pause{
    position: fixed;
    display: none;
    justify-content: center;
    align-items: center;
    flex-direction: column;
    width: 100vw;
    height: 100vh;
    z-index: 1000;
    background-color: rgba(0, 0, 0, .8);
    color: white;
}
```

Вернемся в файл game.js и для класса Game напишем последний необходимый нам параметр, а также вызов метода, который будет отслеживать нажатие клавиши Esc.

```
class Game {
        constructor() {
                this.name = name;
                this.$zone = $('.elements');
                this.elements = [];
                this.player = this.generate(Player);
                this.fruits = [Apple, Banana, Orange];
                this.counterForTimer = 0;
                this.points = 0;
                this.hp = 3;
                this.time = {
                    m1: 0,
                    m2: 0,
                    s1: 0,
                    s2: 0
                };
                this.ended = false;
                this.pause = false;
                this.keyEvents();
        }
```

Опишем поведение метода keyEvents. Будем в нём ожидать нажатие клавиши и менять значение параметра pause на противоположный.

```
keyEvents() {
        addEventListener('keydown', (e) => {
                if (e.key === "Escape") {
                        this.pause = !this.pause;
                }
        })
}
```

Опять необходимо модифицировать метод loop, чтобы теперь он учитывал включена ли пауза и показывал нам её.

```
loop() {
        requestAnimationFrame(() => {
                if (!this.pause) {
                        this.counterForTimer++;
                if (this.counterForTimer % 60 === 0) {
                        this.timer();
                        this.randomFruitGenerate();
                }
                if (this.hp <= 0) {
                        this.end();
                }
                $('.pause').css('display', 'none').hide().fadeOut();
                this.updateElements();
                this.setParams();
                } else if (this.pause) {
                        $('.pause').css('display', 'flex').show().fadeIn();
                }
                if (!this.ended) {
                        this.loop()
                }
        })
}
```

Можете проверять в браузере. Пауза должна работать по клику на клавишу Esc (пример рисунок 9.1).

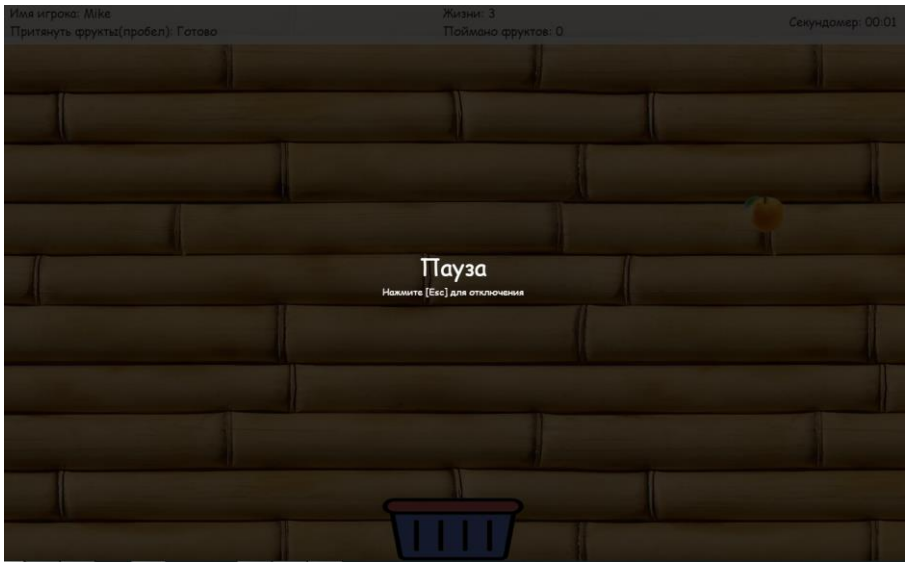

Рисунок 9.1 – Пример работы паузы.

Раздел 10. Разработка логики автоматического сбора фруктов

В этом разделе необходимо вернуться к классу Player. Для начала напишем ещё два параметра. Первый для того, чтобы считать время использования способности, второй для её отката. Также добавим в параметры клавиш «пробел».

```
class Player extends Drawable {
    constructor(game) {
        super(game);
        this.w = 244;
        this.h = 109;
        this.x = this.game.$zone.width() / 2 - this.w / 2;
        this.y = this.game.$zone.height() - this.h;
        this.speedPerFrame = 20;
        this.skillTimer = 0;
        this.couldTimer = 0;
        this.keys = {
            ArrowLeft: false,
            ArrowRight: false,
            Space: false
        };
        this.createElement();
        this.bindKeyEvents();
    }
```

Напишем метод applySkill для класса Player, который будет притягивать фрукты к корзине. В цикле будем перебирать все объекты, кроме корзины и в зависимости от их положения притягивать к середине корзины.

```
applySkill() {
    for (let i = 1; i < this.game.elements.length; i++) {
        if (this.game.elements[i].x < this.x + (this.w / 2)) {
            this.game.elements[i].x += 15;
        } else if (this.game.elements[i].x > this.x + (this.w / 2)) {
            this.game.elements[i].x -= 15;
        }
    }
}
```

Осталось дописать в метод update класса Player проверку условий нажатия клавиши «пробел». Если пользователь отпустит клавишу раньше 4-х секунд, то способность всё равно уйдёт в откат. Также он не сможет её держать дольше 4-х секунд, принудительно запуститься откат способности. Пока будет происходить откат способности, игра не будет реагировать на «пробел».

```
update() {
    if (this.keys.ArrowLeft && this.x > 0) {
        this.offsets.x = -this.speedPerFrame;
    } else if (this.keys.ArrowRight && this.x < this.game.$zone.width() - this.w) {
        this.offsets.x = this.speedPerFrame;
    } else {
        this.offsets.x = 0;
    }
    if (this.keys.Space && this.couldTimer === 0) {
        this.skillTimer++;
        $('#skill').html(`осталось ${Math.ceil((240 - this.skillTimer) / 60)}`);
        this.applySkill();
    }
    if (this.skillTimer > 240 || (!this.keys.Space && this.skillTimer > 1)) {
        this.couldTimer++;
        $('#skill').html(`в откате ещё ${Math.ceil((300 - this.couldTimer) / 60)}`);
        this.keys.Space = false;
    }
    if(this.couldTimer > 300){
        this.couldTimer = 0;
        this.skillTimer = 0;
        $('#skill').html('Готово');
    }
    super.update();
}
```

Поздравляем! Вы смогли реализовать весь необходимый функциональный набор, который необходим для решения задания. Конечно, если вы любите вызов, то можете заглянуть в последний раздел, где вам будут даны задачи для самостоятельной реализации.

Раздел 11. Задания для самостоятельной работы

1) Реализуйте дополнительные анимации игры (переход между экранами, анимация коллизии с корзиной и низом экрана, таймера, траты жизней, прибавления фруктов в счётчик, более сложная анимация отката способности и её использование).
2) Реализуйте логику паузы, в которой будет останавливаться анимация вращения фруктов и возобновляться.
3) Реализуйте правильное поведение способности «Автоматический сбор», который должен собирать все фрукты на экране, включая те, которые почти упали.