**Лаболаторная работа №5**

**Реализация прототипа**

1. **Документация на программный продукт**

* **Архитектурная/проектная документация**

Нашей командой была разработана 2d-игра «Hunter». В ходе разработки мы использовали такие технологии: JavaScript (es6, canvas…), HTML (pug), CSS (SASS), GULP4, GIT, npm/yarn. Таким образом игра написана на чистом JavaScript.

В качестве редактора кода был использован редактор *Visual Studio Code* (причина – один из лучших редакторов для веба на данный момент).

Стиль программирования – ООП (основанное на прототипах – так как игра написана на JavaScript).

Для организации классов(конструкторов) были использованы es6-классы.

* **Техническая документация**

Документация сгенерированная с помощью gulp-jsdoc3: [*http://hunter-2d.tk/doc/*](http://hunter-2d.tk/doc/)

Некоторые фрагменты кода:

*js/game/render.js*

;(function(global){

"use strict"

var startCount = 1; //испльзутеся для корректного старта и паузы

var DEF = {

scale: new Vector(24, 24), //масштаб игры по осям

eachTik: () => {}, //будет выполнятся на каждом кадра (в tik())

}

/\*\*

\* Управление отрисовкой игры

\*

\* @class

\* @param {object} options настройки (по умолчанию DEF)

\*

\* При рендере указываются реальные координаты

\* При работе с классом нужно следить за положением камеры (метод setCamera)

\*

\* Для начала отрисовки необходимо вызвать метод start()

\*/

class Render {

constructor(options = {}) {

this.\_createParametrs(options);

this.\_init();

this.\_initEvents();

}

\_init() {

this.\_getCanvas();

this.\_getCtx();

this.resize();

}

//Запускает отрисовку

//На каждом кадре выполняет метод tik()

start() {

if (this.status.start) return;

this.status.start = true;

var self = this;

var start = startCount;

var time = Date.now();

var prevDilation = 0;

requestAnimationFrame(function tik() {

if (start == startCount) {

var newTime = Date.now();

//замедление времени от отклонения fps

var dilation = Math.min((newTime - time) / 16, 2.5)

//за один кадр замедление не должно

//сильно уменьшаться (но увеличиваться может)

//dilation = Math.max(dilation, prevDilation \* 0.9);

self.tik(dilation); //вызываем в условии, что бы при

//остановки не отрисовывались лишние кадры

time = newTime;

prevDilation = dilation;

requestAnimationFrame(tik);

}

});

}

//Останавливает отрисовку

stop() {

this.status.start = false;

startCount++;

}

/\*\*

\* Функция отрисовки объектов на экране

\*

\* Отисовывает переданные объекты в обратном порятке

\* (то есть первый объект в массиве будет отрисован последним,

\* и следовательно будет находится над другими)

\*

\* Отрисовывает только прямоугольники

\*

\* Также функция не проверяет, будет ли объект виден на экране

\* Это нужно проверять самостоятельно с помщью метода isVisible()

\*

\* @param {array} objects массив отрисовываемых объектов

\* Каждый объект должен иметь вид: {x, y, w, h, img, fillStyle}

\* Если есть свойство img, то fillStyle не используется

\*

\*/

render(objects = []) {

var ctx = this.ctx;

//проверка количества отрисовываемых объектов

//console.log(objects.length);

for (var i = objects.length - 1; i >= 0; i--) {

var obj = this.\_getObjOnScreen(objects[i]);

this.ctx.save();

if (objects[i].img) {

ctx.drawImage(objects[i].img, obj.x^0, obj.y^0, obj.w^0, obj.h^0);

} else {

ctx.fillStyle = objects[i].fillStyle || 'rgba(0,0,0,0.3)';

ctx.fillRect(obj.x^0, obj.y^0, obj.w^0, obj.h^0);

}

this.ctx.restore();

}

}

//Кадр анимации

tik(dilation) {

if (!this.status.start) return;

this.clear();

this.options.eachTik(dilation);

}

/\*\*

\* Установить положение (координаты камеры)

\* @param {Vector} camera - вектор, сорержащий реальные координаты камеры

\*/

setCamera(camera) {

this.camera = camera.scale(this.options.scale);

}

//Переопределение размеров игрового

resize() {

this.updateMetrics();

this.canv.width = this.metrics.gameSize.x;

this.canv.height = this.metrics.gameSize.y;

//Для отключения сглаживания (очень важно)

global.setCanvasSmoothing(this.ctx, false);

}

//Обновляет размеры используемые для расчетов

updateMetrics() {

var m = this.metrics;

m.gameSize = new Vector(

window.innerWidth,

window.innerHeight

);

m.realGameSize = m.gameSize.diScale(this.options.scale);

}

//Очищает игровое поле

clear() {

this.ctx.clearRect(0, 0, this.metrics.gameSize.x, this.metrics.gameSize.y);

}

/\*\*

\* По передонному объекту с реальными коондинатами

\* Возвращает объект координат и размеров прямоугольника на экране

\*

\* @param {object} obj

\*/

\_getObjOnScreen(obj) {

var scale = this.options.scale;

return {

x: (obj.x \* scale.x) - this.camera.x + this.metrics.gameSize.x / 2,

y: (obj.y \* scale.y) - this.camera.y + this.metrics.gameSize.y / 2,

w: obj.w \* scale.x,

h: obj.h \* scale.y,

}

}

/\*\*

\* Проверяет, будет ли отображатся переданным объект на экране

\*

\*

\* @param {object} realObj - объект вида {x, y, w, h} -

\* все значения - реальные

\*

\* @param {number} k - изменение реальной области экрана

\* если k = 0, то будут видимыми будут считаться только те объекты,

\* которые действительно будут видны

\*

\* если k > 0, то область будет увеличина в k раз

\* если k < 0, то область будет уменьшина в k раз

\*/

isVisible(realObj, k = 0.5) {

var obj = this.\_getObjOnScreen(realObj);

//делаем отдельно для k = 0, что бы при таком k

//производительность метода была максимальной

if (k === 0) {

return isIntersectRect(

0, this.metrics.gameSize.x, 0, this.metrics.gameSize.y,

obj.x, obj.x + obj.w, obj.y, obj.y + obj.h

);

}

var gw = this.metrics.gameSize.x;

var gh = this.metrics.gameSize.y;

//Дополнительные размеры области

k /= 2;

var kw = gw \* k;

var kh = gh \* k;

return intersectRect(

-kw, gw + kw, -kh, gh + kh,

obj.x, obj.x + obj.w, obj.y, obj.y + obj.h

);

}

\_getCtx() {

this.ctx = this.canv.getContext('2d');

}

\_getCanvas() {

this.canv = document.querySelector('.game\_\_main\_canv');

}

//Иницилизирует необходимые DOM-события

\_initEvents() {

var resizeTimer = 0;

global.addEventListener('resize', () => {

clearInterval(resizeTimer);

resizeTimer = setTimeout(() => {

requestAnimationFrame(() => {

this.resize();

this.tik();

})

}, 16);

});

global.addEventListener('load', () => {

this.resize();

this.tik();

});

}

//Создает основные параметры испльзуемые конструктором

//options - передаются из конструктора при иницилизации

\_createParametrs(options) {

this.options = extend(true, {}, DEF, options);

this.canv = null;

this.ctx = null;

this.camera = new Vector(0, 0);

this.metrics = {};

this.status = {

start: false,

}

}

}

global.Game.Render = Render;

}(window));

*js/game/sprite.js*

;(function(global){

"use strict"

//Кеширование tileset-ов

//Содержит поля у который ключ - это путь к картинке

//А значение - либо DOM элемент tileset-a (Image)

//Либо null если картинку с таким адресом загрузить не удалось

var loadedTilests = {}

var DEF = {

tileset: 'tileset.png', //Путь к тайлсету со спрайтами

size: new Vector(32, 32), //Размеры элемента канваса/спрайта

//Объект, сореджащий кадры анимаций по типам

//Тип передается при вызове метода start

//И испльзуется для выбора кадров, которые стоит отрисовывать

cadrs: {

//Каждый тип - этом массив, содержащий объекты вида:

base: [{

metrics: {

//x, y - точки на tileset-e с которых начинается кадр

//w, h - ширина и высота тайла

x: 0,

y: 0,

w: 32,

h: 32,

},

mirrorX: false, //Отзеркаливать по горизонтали (кадр)

mirrorY: false, //Отзеркаливать по вертикали (кадр)

//Длительность показа данного кадра

duration: 0,

}],

},

mirrorX: false, //Отзеркаливать по горизонтали (все кадры)

mirrorY: false, //Отзеркаливать по вертикали (все кадры)

}

/\*\*

\* Класс для работы со спрайтами и их анимациями

\*

\* Конструктор получает объект настроек в формате DEF

\*

\* Порядок работы:

\* - содание екземпляра

\* - вызов метода start(type), когда нужно начать анимацию нужного типа

\* - вызов метода stop(), когда нужно остановить анимацию

\*

\* - - - - -

\* Класс Sprite наследует от Events

\* Поэтому у него есть методы события

\* (addEvent, trigger...)

\*

\* Поддерживаемые события:

\* "tileset\_load" - вызывается, после загрузки тайлсета

\* "before\_chande\_cadr" - вызывается перед смене кадра

\*

\* @class

\*/

class Sprite extends Events {

constructor(options = {}) {

super();

this.\_createParametrs(options);

this.\_init();

}

\_init() {

this.\_loadTileset().then(() => {

this.\_createCanvas();

this.\_setSize();

this.trigger('tileset\_load', this);

}, () => {

});

}

/\*\*

\* Начинает анимацию спрайта переданного типа

\*

\* @param {string} type типа анимации

\* @param {bool} requiredStart запустить в любом случае

\*

\* Праметры типа - options.cadrs[type]

\* Параметры должны содержать параметры хотя бы одного кадра

\*

\* Пока не загружен tileste ничего не делает.

\*/

start(type, requiredStart) {

if (this.tileset === null || this.ctx === null) return;

if (!type) {

console.error(

'Неправильный тип анимации (', type,

'). Будет использован тип "base"'

);

type = 'base';

}

//Таже анимация

if (type === this.curAnimateType) {

return; //убирает возможность многократного вызова аниации

//одного вида, из-за чего может зациклится первый кадр

}

this.curAnimateType = type;

//Если какая-то анимация уже запущена

if (this.timer != 0) {

this.stop();

}

var config = this.options.cadrs[type];

if (!config || config.length === 0) {

console.error('Нет кадров в анимации типа', type);

return;

}

if (config.length === 1) {

this.\_drawNext(config[0]);

return;

}

var self = this;

var cadrIndex = 0;

this.\_drawNext(config[cadrIndex]);

this.timer = setTimeout(function timerFunc() {

cadrIndex = (cadrIndex + 1) % config.length;

self.\_drawNext(config[cadrIndex]);

self.timer = setTimeout(timerFunc, config[cadrIndex].duration);

}, config[cadrIndex].duration);

}

/\*\*

\* Отрисовывает кадр, с переданными параметрами

\* Выполняя все необходыми дополнительные действия

\* (непосредственно рисует, и вызвает события)

\*

\* @param {object} config

\*/

\_drawNext(config) {

var allow = true; //разрешение на рисование кадра

//Вызывается событие "before\_chande\_cadr"

//В hendlers передается 2 параметра

//Первый - параметры кадра

//Вротой - фукнция, для запрета отрисовки кадра

//Для этого ее нужно вызвать:

//sprite.addEvent('before\_chande\_cadr', (config, disallow) => {

// disallow();

//});

this.trigger('before\_chande\_cadr', config, function disallow() {

allow = false;

});

if (allow) {

this.\_draw(config.metrics, config.transforms);

}

};

//Останавливает анимацию

stop() {

if (this.timer === 0) return;

clearTimeout(this.timer);

this.timer = 0;

this.curAnimateType = '';

}

/\*\*

\* Меняет размеры канваса/спрайта

\*

\* @param {Vector} newSize новый размер

\*

\* Все что нарисовано на спрайте будет стерто

\*/

changeSize(newSize) {

this.options.size = newSize;

this.\_setSize();

}

/\*\*

\* Загружает tileset по адресу из options.tileset

\*

\* Кеширует tileset-ы, которые удалось и не удалось загрузить

\*

\* Возвращает промис

\*/

\_loadTileset() {

var src = this.options.tileset;

return new Promise((resolve, reject) => {

//Проверка закешированных tileset-ов

var loaded = loadedTilests[src];

//Закешированно, но не было загруженно

if (loaded === null) {

reject();

return;

//Закешированно и удачно загруженно

} else if (loaded instanceof Image) {

this.tileset = loaded;

resolve();

return;

}

var tileset = new Image();

tileset.onload = () => {

loadedTilests[src] = tileset; //кешируем tilest

this.tileset = tileset;

resolve();

}

tileset.onerror = () => {

//Проверка обязательно должно произмодиться не через

//Логическое преобразование(так как в проверяемом объекте

//Могут содержатсья занчения null, которые тоже нужно учитывть)

if ( !(src in loadedTilests) ) {

console.error(

'Не удалось загрузить tileset: "',

this.options.tileset, '"'

);

loadedTilests[src] = null; //кешируем, что такого tileset-a нет

}

reject();

}

tileset.src = src;

});

}

/\*\*

\* Рисует кадр спрайта

\*

\* @param {Object} metrics положение и размеры спрайта в tileset-e

\* @param {Vector} transform трансформации (формат из options.cadr)

\*

\* Очищает холст перед рисованием очищает

\*/

\_draw(metrics, transforms) {

var config = this.\_getDrawConfig(metrics, transforms);

if (this.\_isSameDrawConfig(config, this.\_prevDrawCofig)) {

return;

}

this.\_prevDrawCofig = config;

this.\_clear();

if (!this.ctx) console.log(this.canvas)

this.ctx.save();

this.\_setMirror(config);

this.ctx.drawImage(

this.tileset,

config.x^0,

config.y^0,

config.w^0,

config.h^0,

0,

0,

this.options.size.x^0,

this.options.size.y^0

);

this.ctx.restore();

}

/\*\*

\* Возвращает параметры кадра

\*

\* @param {object} metrics метрики кадра

\* @param {object} transforms трансформации кадра

\*/

\_getDrawConfig(metrics, transforms = {}) {

return {

x: metrics.x,

y: metrics.y,

w: metrics.w,

h: metrics.h,

mirrorX: transforms.mirrorX ^ this.options.mirrorX,

mirrorY: transforms.mirrorY ^ this.options.mirrorY,

}

/\*

Смысл использования XOR в даной фукцнии:

Отзеркаливание в спрайтах можно применять непоследственно для

кадра, а также для всех кадров.

При этом отзеркаливания налаживаются.

Если кадр не отзеркален, при глобальном отзеркаливании

он будет отзеркален, а если отзеркален, то он будет отзеркален

дважды - то есть он не будт отзеркален.

Проанализировав можно сделать вывод, что это и есть XOR

Если разные - то отзеркаливаем. Если одинаковые - то нет

\*/

}

/\*\*

\* Возвращает одинаковые ли переданные drawConfig-и

\*

\* @param {object} config1 первый drawConfig

\* @param {object} config2 второй drawConfig

\*/

\_isSameDrawConfig(config1, config2) {

if (!config1 || !config2) return;

return (

config1.x === config2.x &&

config1.y === config2.y &&

config1.w === config2.w &&

config1.h === config2.h &&

config1.mirrorX === config2.mirrorX &&

config1.mirrorY === config2.mirrorY

);

}

/\*\*

\* Устанавливает отзеркаливание кадра

\*

\* @param {object} config drawConfig кадра

\*/

\_setMirror(config) {

if (config.mirrorX) {

this.ctx.translate(this.options.size.x, 0);

this.ctx.scale(-1, 1);

}

if (config.mirrorY) {

this.ctx.translate(0, this.options.size.y);

this.ctx.scale(1, -1);

}

}

//Очищает канвас спрайта

\_clear() {

if (!this.ctx) return;

this.ctx.clearRect(

0,

0,

this.options.size.x,

this.options.size.y

);

}

\_setSize() {

if (!this.sprite) return;

this.sprite.width = this.options.size.x;

this.sprite.height = this.options.size.y;

global.setCanvasSmoothing(this.ctx, false);

}

\_createCanvas() {

this.sprite = document.createElement('canvas');

this.ctx = this.sprite.getContext('2d');

}

\_createParametrs(options) {

this.options = extend(true, {}, DEF, options);

this.timer = 0; //Таймер, используемый для setTimeout

this.tileset = null;

this.sprite = null; //canvas содержащий спрайт

this.ctx = null;

this.curAnimateType = ''; //Текущий тип анимации

//Параметры последнего отрисованного кадра

//Используется, что бы не отрисовывать подряд одинковые кадры

this.prevDrawCofig = {

metrics: {},

transforms: {},

}

}

}

global.Game.Sprite = Sprite;

}(window));

js/game/objects/block.js

;(function(global){

"use strict"

var DEF = {

//Метрики блока

x: 0,

y: 0,

w: 1,

h: 1,

//Физические параметры блока

props: {

damage: 0,

armor: 0,

hurtInterval: 500,

},

fillStyle: '', //Цвет, что будет использоваться, когда нет картинки

//Настройки для спрайтов передаваемые в new Sprite

//Должен содержать массив объектов кадров в формета

//Указанном в Game.Sprite

//Непосредственно для Block-ов будет записан

//В тип кадров 'base' (cadrs.base)

animation: [{

metrics: {

x: 0,

y: 0,

w: 16,

h: 16,

},

duration: 0,

}],

tileset: '', //Адресс к tileset-у

tileW: 16,

tileH: 16,

}

/\*\*

\* Класс для создания игрового блока

\*

\* @class

\*/

class Block extends Game.Rect {

/\*\*

\* @param {object} options опции

\* @param {Collisions} collisions - collisions

\*/

constructor(options = {}, collisions) {

super();

this.collisions = collisions;

this.\_createParametrs(options);

this.\_init();

}

hurt(obj, startDamage = 0) {

var time = Date.now();

if (time - this.\_prevHurt < this.\_hurtInterval) {

return;

}

var damage = startDamage + this.damage;

obj.pain(damage);

this.\_prevHurt = time;

}

pain() {

}

/\*\*

\* Включает все процессы, которые выполняются периодично

\* (отрисовку спрайта)

\*/

start() {

if (!this.sprite) this.\_initSprite();

this.sprite.start('base');

}

/\*\*

\* Выключает все процессы, которые выполняются периодично

\* (отрисовку спрайта)

\*/

stop() {

if (this.sprite) this.sprite.stop();

}

/\*\*

\* Возвращает объект параметров для рендера

\*/

convertToRender() {

return {

x: this.coords.x,

y: this.coords.y,

w: this.size.x,

h: this.size.y,

img: (this.sprite) ? this.sprite.sprite : null,

fillStyle: this.options.fillStyle,

}

}

\_init() {

}

\_initSprite() {

var o = this.options;

this.sprite = new Game.Sprite({

tileset: o.tileset,

size: new Vector(o.tileW, o.tileH),

cadrs: {

base: this.options.animation,

},

});

}

updateCoords() {

//Для Block-а ничего не делает

}

tik() {

}

clearMoveStatus() {

}

/\*\*

\* Реакия на взаимодействие с другим объектом

\* (изменяет параметры переданного объека)

\*

\* Обеспечивает такие эффекти на переданный обект:

\* - нормальная реакция опоры

\* - сила трения

\* - нанесение урона

\* - и др.

\*

\* @param {Block} obj - другой объект (наследующий от Block)

\* (на объекты типа block не влияет

\* влияет на объекты dinamicBlock и их наследников)

\*

\* @param {Block} side - cторона взаимодействия (left, top, right, bottom)

\*/

respondInteraction(obj, side) {

//this.speed = this.speed.plus(obj.speed.mul(-0.3));

var startDamage = 0;

//Если это именно блок

if (obj.type === 'block') return;

if (side === 'top') {

startDamage += this.\_convertSpeedToPain(obj.speed.y);

obj.speed.y = 0;

obj.bottom = this.top;

} else if (side === 'bottom') {

startDamage += this.\_convertSpeedToPain(obj.speed.y);

obj.speed.y = 0;

obj.top = this.bottom;

} else if (side === 'left') {

startDamage += this.\_convertSpeedToPain(obj.speed.x);

obj.speed.x = 0;

obj.right = this.left;

} else if (side === 'right') {

startDamage += this.\_convertSpeedToPain(obj.speed.x);

obj.speed.x = 0;

obj.left = this.right;

}

this.hurt(obj, startDamage);

//Если это персонаж или его наследники

if (obj instanceof Game.Actor) {

if (side === 'top') {

obj.jump();

}

if (side === 'top' || side === 'bottom') {

obj.speed.x /= 10;

}

}

obj.afterCollision();

}

afterCollision() {

}

\_convertSpeedToPain(speed) {

return Math.pow(Math.abs(speed \* 5), 4);

}

\_createParametrs(options) {

this.options = extend(true, {}, DEF, options);

this.coords.x = this.options.x;

this.coords.y = this.options.y;

this.size.x = this.options.w;

this.size.y = this.options.h;

this.speed = new Vector(0, 0);

this.sprite = null;

this.damage = this.options.props.damage;

this.armor = this.options.props.armor;

this.\_hurtInterval = this.options.props.hurtInterval;

this.\_prevHurt = 0;

}

}

Object.defineProperty(Block.prototype, 'type', {

value: 'block',

});

global.Game.Block = Block;

}(window));

* **Пользовательская документация (руководство пользователя)**

Руководство пользователя выполнено на основании ГОСТ 19.505-79, ГОСТ 19.105-77, ГОСТ 19.105-78.

1. Назначение программы

Функциональное назначение

Функциональным назначением программы является предоставление пользователю возможности игры в созданную игру для развлечения и досуга.

Эксплуатационное назначение

Игра предназначена для всех пользователей, которые желают поиграть на досуге. Для того, чтобы играть в игру пользователю не нужно ничего кроме устройства, на котором он будет играть и интернета.

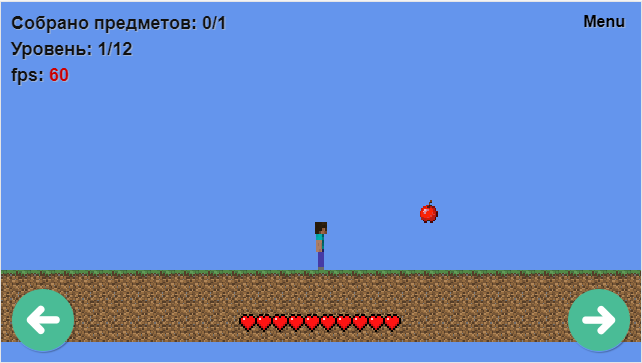
Поддерживаемые браузеры: все современные браузеры начиная с *InernetExplorer 9.*

Разрешение экрана – разрешение всех современных устройств

Операционная система – любая, для запуска игры нужен только браузер. А играть можно как на компьютере, так и на телефоне и планшете.

1. Запуск игры

Чтобы начать игру нужно открыть страницу в браузере: [*http://hunter-2d.tk*](http://hunter-2d.tk). Играть можно сразу после загрузки страницы.



1. Управление:

В игре есть несколько способов управления:

* A/D/W (налево, направо, прыжок)
* Клавише со стрелками
* Касание по кнопкам - налево, направо, касание по экрану - прыжок (для сенсорных экранов)

Во время прыжка игрок имеет возможность менять направление.

Прыгать, также, можно на пробел.

1. Описание игры:

Цель игры состоит в том, чтобы собрать все ценные предметы, и после этого прийти на финиш.

Если прийти на финиш, до того, что будут собраны все ценные предметы, то будет выведено соответствующее уведомление.

После победы на уровне загружается следующий уровень. Нужно победить на всех уровнях.

Ценными предметами, являются: яблоко, золотое яблоко, алмаз, изумруд, и другие предметы (в зависимости от уровня).

Ценные предметы анимируются (подымаются верх, вниз).

Пример ценных предметов:

Финиш:



У игрока есть жизни:



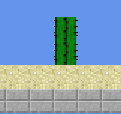
При падении с большой высоты, или при контакте с опасными объектами они уменьшаются:



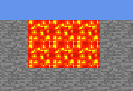
Когда жизни заканчиваются – игрок уминает. При этом перезапускается текущий уровень.

До опасных предметов принадлежат:

Кактус:



Лава:



1. Меню

При нажатии на кнопку «меню», либо на нажатии на клавишу **Еsc** открывается меню:



При открытии меню игра останавливается.

Если текущий уровень – не первый, то при выборе «Начать уровень сначала» будет запрошено подтверждение.

1. Сохранение

Игра автоматически сохраняет текущий уровень. Таким образом при повторном открытии страницы будет открываться последний уровень, на котором находился игрок.

**Разработка и отладка кода программы**

В ходе работы над проектом нашей командой был разработан и отлажен программный код. Для отладки мы использовали встроенные в браузеры параметры разработчиков. Во время разработки основным браузером был Яндекс.Браузер и Хром.

**Разработка плана тестирования программы и проведения тестирования**

**План тестирования программы**

В ходе этапа тестирования планируется протестировать 2d-игру «Hunter».

В ходе тестирования необходимо протестировать все фрагменты исходных кодов, а также протестировать работу игры в разных браузерах и при разных разрешениях.

Нашей командой было принято решение проводить следующие виды тестирования: автоматическое тестирования BDD, тестирование интерфейса, бета-тестирование.

Автоматическое тестирование проводится параллельно с написание кода согласно методологи BDD.

Бета-тестирование начинается после реализации всех основных функций игры.

Критерием окончания тестирования является состояние программного продукта, при котором все найденные ошибки были устранены, и при котором он стал удовлетворять требования ТЗ и критерии качества.

**BDD**

BDD – разработка через тестирование. Суть – написание юнит-тестов перед написанием кода. Это обеспечивает высокий уровень планирования и осознания назначения реализуемых фрагментов кода, а также позволяет избежать большого количество ошибок.

Тестирование было организовано с помощью библиотек: mocha.js и chai.js.

Страница с тестами: <http://hunter-2d.tk/test/>

В ходе автоматического тестирования было обнаружено и исправлено большое количество разнообразных ошибок.

Тестирование в различных браузерах и при различных разрешениях:

При тестировании кроссбраузерности были выявлены серьёзные проблемы с ИЕ9. Но благодаря их обнаружению они были устраненны.

При тестировании на разных разрешение было достигнуто адекватное отображение игры на практически любых разрешениях (вплоть до 250х250 пикселе). При тестировании было проверено отображение на всех основных разрешениях (для смартфонов, планшетов, ноутбуков…)

**Бета-тестирование**

Бета-тестирование – это тестирование продукта среди группы добровольцев.

Для бета-тестирование игра была предложена некоторым группам потенциальных пользователей. В результате тестирования были выявлены и исправлены ошибки:

1. Фокус на кнопках меню, при его закрытии, из-за чего после нажатия «начать уровень сначала» при каждом нажатии пробела этот пункт меню выбирался снова.
2. Лава иногда снимала жизни тогда, когда игрой ее не касался.
3. По рекомендациям пользователей была добавлена пауза при открытии меню.