

Технически Университет – Варна

Проект
по “Графични Системи”

Задание: Реализация на анимация на подскачащ обект съобразно управлението от мишката

Съдържание

- 1.Описание на проекта
- 2.Преглед на предметната област
- 3.Проектиране
- 4.Реализация
- 5.Демонстрационни кадри
- 6.Заклучение и използвани източници

1.Описание на проекта

Целта на проекта е да се състави програма реализираща анимация на подскачащ обект съобразно управлението от мишката под формата на компютърна игра. За вдъхновение представлява играта „Jump King“ на Nexile, Ukiyo Publishing.

Играта се състои от едно ниво, разделено на 3 участъка. Играчът управлява движенията на героя в центъра на екрана. Движенията се осъществяват чрез насочени подскоци в посока, определена от местоположението на курсора спрямо героя. Подскоците се осъществяват чрез задържане левия бутон на мишката и пускане след определен период. Продължителността на задържането определя силата на подскока.

Целта на играча е достигането до „звезда“ намираща се на края на нивото, събирайки възможно най-голям брой „монети“ носещи точки. В нивото също се намират и „знамена“, които запазват напредъка на играча. Като препядствия са разпръснати „триони“. При контакт с тях героя се връща в началното положение или последно достигнатото знаме, а играчът губи една точка.

Трудността на играта, определена от разположението на платформите и препядствията, се увеличава в зависимост от напредъка на играча в нивото.

2.Преглед на предметната област

В проекта са използвани формули и техники от компютърната графика като трансляция и ротация на точка. Използвани са вградени функции за изчертаване и запълване многоъгълник с цвят, текстура и градиент. Налице са и оптимизации върху производителността чрез определяне на обекти намиращи се извън екрана. Използван е и алгоритъм за откриване на сблъсъци между обекти.

3.Проектиране

Като средства за реализация са избрани езиците за програмиране *HTML*, *CSS* и *Javascript*. Проекта съдържа един HTML файл *index.html*, който указва и повиква скриптовите. Скриптовите файлове се намират в папката *scripts*, а изображенията използвани като текстури в папката *assets*. Всички класове в проекта и повечето функции се намират в собствени файлове, с изключение на *math.js*, който съдържа полезни функции за графични

изчисления, и *input.js* , който съдържа функции отчитащи управлението от мишката чрез *Event Listener*.

Класовете *Cloud*, *Coin*, *Player*, *SavePoint*, *Saw*, *Star* и *Wall* представляват игровите обекти, които се визуализират на екрана. Всички те наследяват от класа *Box*, описващ правоъгълник, който съдържа атрибути за местоположение и размери. Чрез неговите методи може да се разбере дали даден обект се припокрива с друг. Класът *Box* от своя страна наследява от класа *Vector* съдържащ координати *x* и *y*.

Всички игрови обекти се съхраняват в различни масиви според тяхната роля във файла „*game_objects.js*“. Техните координати и стойности са зададени в кода. Игровата логика както и кода за графиката се контролира и изпълнява от функцията *main* в „*main.js*“. Тя се извиква 60 пъти за секунда, осигурявайки опресняване на екрана 60 пъти в секунда. Всички игрови обекти се визуализират спрямо дадена точка (*focusPoint*), която винаги приема координатите на центъра на героя.

4.Реализация

Класът *Box* се наследява от повечето игрови обекти и представлява *Bounding box* на конкретния обект. Чрез метода *overlap* се проверява дали обекта се намира на екрана, след подаване на местоположението и размера на екрана под формата на друга инстанция на класа. Ако се припокриват тогава знаем, че обекта се намира на екрана и трябва да бъде нарисован.

Методът *fix* връща минималните стойности за изменение на координатите нужни, така че обекта да не се припокрива със зададения в параметъра. Използва се при сблъсък на героя(*Player*) със платформа/стена(*Wall*). Методът работи като проверява на какво разстояние героя на навлязал в обекта за всяка от четирите ѝ страни. Избират се минималните стойности за *x* и *y* съответно от ляво/дясно и горе/долу. Получените стойности се връщат като резултат.

```
class Box extends Vector{

  constructor(x, y, width, height){
    super(x,y);
    this.width = width;
    this.height = height;
  }

  overlap(box){
    return !(
      (this.x+this.width<box.x) || (this.x>box.x+box.width) ||
      (this.y+this.height<box.y) || (this.y>box.y+box.height));
  }

  fix(box){
    let distLeft = this.x+this.width - box.x;
    if(distLeft<0 || distLeft>box.width/2)
      distLeft = 0;

    let distRight = - ( this.x - ( box.x + box.width));
    if(distRight<0 || distRight>box.width/2)
      distRight = 0;

    let distTop = this.y + this.height - box.y;
    if(distTop<0 || distTop>box.height/2)
```

```

        distTop = 0;

        let distBottom = - ( this.y - (box.y+box.height));
        if(distBottom<0 || distBottom>box.height/2)
            distBottom = 0;

        if(distLeft == 0 && distRight == 0 && distTop == 0 && distBottom == 0)
            return new Vector(0,0);

        if(smallestNonZero(distLeft, distRight, distBottom, distTop)){//go left
            console.log("going left:",distLeft);
            return new Vector(-(distLeft+1), 0);
        }else if(smallestNonZero(distRight, distLeft, distBottom, distTop)){//go
right
            console.log("going right:",distRight);
            return new Vector(distRight+1, 0);
        }else if(smallestNonZero(distBottom, distLeft, distRight, distTop)){//go
down
            console.log("going down:",distBottom);
            return new Vector(0, distBottom+1);
        }else{//go up
            console.log("going up:",distTop);
            return new Vector(0, -(distTop+1));
        }
    }
}

```

Класът *Wall* е отговорен за съхранение на координатите и размерите на стените и платформите, както и тяхната визуализация. Атрибутът *pattern* представлява текстурата с която се запълва обекта. Тексурите се създават от изображения и се подават като параметър на конструктора във файла *game_objects.js* , където също се създават и обекти от класа *Wall*.

Методът *draw* е отговорен за визуализацията на обекта на екрана. В него се изчислява отместването спрямо x и y (*offsetX*, *offsetY*) в зависимост от местоположението на *focusPoint* и размерите на *canvas* елемента. Чрез метода *setTransform* се извършва транслация на контекста в зависимост от отместването. По този начин също се отмества и текстурата. След което се начертава правоъгълник с размери, определени от атрибутите на обекта и се запълва с текстурата. Накрая отново чрез *setTransform* се възстановява състоянието контекста, като се зададат нулеви стойности за отместването.

```

class Wall extends Box{
    constructor(x, y, width, height, pattern){
        super(x,y,width,height);
        this.pattern = pattern;
    }

    draw(ctx, canvas, relativeTo){
        const offsetX = canvas.width/2 - relativeTo.x;
        const offsetY = canvas.height/2 - relativeTo.y;

        ctx.beginPath();
        ctx.lineWidth = 3;
        ctx.strokeStyle = "rgb(0,0,0)";
        ctx.fillStyle = this.pattern;
    }
}

```

```

        ctx.setTransform( 1, 0, 0, 1,
            this.x + offsetX,
            this.y + offsetY);

        ctx.fillRect(0, 0,this.width,this.height);
        ctx.rect(0,0,this.width, this.height);
        ctx.stroke();
        ctx.setTransform( 1, 0, 0, 1, 0, 0 );
    }
}

```

Класът *Coin* е отговорен за съхранение на координатите на монетите, както и тяхната визуализация. Номера на текущия кадър от анимацията се съхранява в *this.animationFrame*, а на последния в статичната променлива *COIN_LAST_FRAME_N*.

Методът *draw* визуализира обекта, представляващ елипса. Стойността на радиусът *x* (*radiusX*) на елипсата се изчислява в зависимост от номера на текущия кадър от анимацията. Стойността на текущия кадър се увеличава при всяко извикване на метода, но ако тя надвишава номера на последния кадър (*static COIN_LAST_FRAME_N*) се връща в началната си стойност – 0. В резултат се създава анимация на стесняване или разширяване на елипсата, наподобявайки въртенето на монета.

```

class Coin extends Box{
    static COIN_LAST_FRAME_N = 100;
    constructor(x,y){
        super(x,y,30,30);
        this.animationFrame = 0;
    }

    draw(ctx, canvas, relativeTo){
        const radiusX = this.width * Math.abs((Coin.COIN_LAST_FRAME_N/2 -
this.animationFrame)/(Coin.COIN_LAST_FRAME_N/2));
        ctx.beginPath();
        ctx.lineWidth = 1;
        ctx.strokeStyle = "rgb(0,0,0)";
        ctx.setTransform( 1, 0, 0, 1,
            this.x - relativeTo.x + canvas.width/2,
            this.y - relativeTo.y + canvas.height/2);
        ctx.fillStyle = "rgb(255,255,0)";
        ctx.ellipse(this.width/2,this.height/2,radiusX, this.height, 0, 0,
2*Math.PI);
        ctx.fill();
        ctx.stroke();
        ctx.setTransform( 1, 0, 0, 1, 0, 0 );
        this.animationFrame += 1;
        if(this.animationFrame>Coin.COIN_LAST_FRAME_N){
            this.animationFrame = 0;
        }
    }
}

```

Класът *Player* е отговорен за съхранение на координатите и скоростта на героя, както и осъществяването на неговото движение в зависимост от скоростта. Скоростта се увеличава при подскок и бавно се намалява в оста *y* (под въздействието на гравитацията) и още по-бавно в оста *x* (под въздействието на въздушното съпротивление). При сблъсък със стена

скоростта спрямо осите x или y приема стойност 0, в зависимост от посоката на сблъсъка. Класът съдържа и инстанция на класа *Trail*, който запазва предишните координати на обекта. Използва се за изчертаване на пунктирна линия, показваща движението на обекта при скок.

Методът *simulate* е отговорен за движението на героя спрямо неговата скорост и модифициране на скоростта в зависимост от земното притегляне и съпротивлението на въздуха. Извиква се от функцията *main* на всяка стъпка от симулацията (60 пъти в секунда).

Методът *hit* е отговорен за осъществяването на сблъсъци между героя и стените. Той измества местоположението на обекта в зависимост от резултата на метода *fit* и променя скоростта в зависимост от посоката на сблъсъка.

Методът *draw* е отговорен за визуализиране на героя на екрана. Той присъства при всички игрови обекти които се виждат на екрана, но е с различна имплементация. Като параметри приема HTML елемента *canvas*, 2D контекстът му и точката която се намира в центъра на екрана. В тялото на метода се изчислява и запазва отместването спрямо x и y (*offsetX*, *offsetY*) в зависимост от местоположението на *focusPoint* и размерите на *canvas* елемента. Отместването се използва за преобразуване от координатите на обекта в нивото в координати на обекта на екрана. В следващата стъпка се създават точките дефиниращи героя, в зависимост от височината и размера му. След което тези точки се трансформират чрез трансляция в зависимост от набраната сила за скок (*this.jump*), определена от продължителността на задържане на левия бутон на мишката. Извършва се и ротация спрямо точката в основата на героя в зависимост от местоположението на мишката. Накрая точките се свързват с линии.

```
class Player extends Box{
    static MAX_VELOCITY = 20;
    static AIR_RESISTENCE = 0.05;
    static GRAVITY_STRENGTH = 0.5;
    static JUMP_INCREASE = 0.03;
    static MAX_JUMP = 50;

    constructor(x, y, width, height){
        ...
    }
    ...

    draw(ctx, canvas, relativeTo){
        const offsetX = canvas.width/2 - relativeTo.x;
        const offsetY = canvas.height/2 - relativeTo.y;
        //points forming the character
        const R = new Vector(this.x+this.width/2+offsetX,
this.y+7*this.height/8+offsetY);

        let A = new Vector(this.x + this.width/2 + offsetX, this.y + this.height/3 +
offsetY);
        let B = new Vector(this.x + this.width/2 + offsetX, this.y + 2*this.height/3 +
offsetY);
        let C = new Vector(this.x + this.width/8 + offsetX, this.y + 5*this.height/8 +
offsetY);
        let D = new Vector(this.x + 7*this.width/8 + offsetX, this.y + 5*this.height/8
+ offsetY);
        let E = new Vector(this.x + this.width/4 + offsetX, this.y + 3*this.height/4 +
offsetY);
        let F = new Vector(this.x + this.width/4 + offsetX, this.y + this.height +
offsetY);
```

```

    let G = new Vector(this.x + 3*this.width/4 + offsetX, this.y + 3*this.height/4
+ offsetY);
    let H = new Vector(this.x + 3*this.width/4 + offsetX, this.y + this.height +
offsetY);
    let I = new Vector(this.x + this.width/2 + offsetX, this.y + this.height/5 +
offsetY);
    //translation
    A.add(0, this.jump*this.height/6);
    B.add(0, this.jump*this.height/6);
    C.add(-this.jump*this.width/4, -this.jump*this.height/8);
    D.add(this.jump*this.width/4, -this.jump*this.height/8);
    E.add(-this.jump*this.width/12, this.jump*this.height/12);
    G.add(this.jump*this.width/12, this.jump*this.height/12);
    I.add(0, this.jump*this.height/6);
    //find angle
    let angle = 0;
    if(!this.falling){
        if(isMouseDown){
            const direction = new Vector(mousePos.x-canvas.width/2, canvas.height/2
- mousePos.y);

            let tempAngle = angleX(direction);

            //angle must be between 0 and PI
            if(tempAngle > 3*Math.PI/2){//down right
                tempAngle = 0;
            }else if(tempAngle > Math.PI){//down left
                tempAngle = Math.PI;
            }

            angle = 2*Math.PI - ( tempAngle - Math.PI/2);

            console.log("direction=",direction," angle=",angle);
        }else{
            angle = 0;//point up
        }

        this.lastAngle = angle;
        this.trail.reset();
    }else{
        angle = this.lastAngle;
        this.trail.addPoint(new Vector(this.x+this.width/2, this.y+this.height));
        this.trail.draw(ctx, canvas, relativeTo);
    }

    //points after rotation

    A = rotatePoint(A, R, angle);

    . . .
    I = rotatePoint(I, R, angle);

    //connecting the points
    ctx.lineWidth = 4;
    ctx.strokeStyle = "rgb(0,0,0)";
    ctx.fillStyle = "rgb(0,0,0)";
    ctx.beginPath();
    ctx.arc(I.x, I.y, player.width/4, 0, 2*Math.PI);
    ctx.fill();
    ctx.moveTo(A.x, A.y);
    ctx.lineTo(B.x, B.y);

```

```

    ctx.lineTo(E.x, E.y);
    ctx.lineTo(F.x, F.y);
    ctx.moveTo(B.x, B.y);
    ctx.lineTo(G.x, G.y);
    ctx.lineTo(H.x, H.y);
    ctx.moveTo(C.x, C.y);
    ctx.lineTo(A.x, A.y);
    ctx.lineTo(D.x, D.y);
    ctx.stroke();
}

simulate(){
    if(this.falling){
        this.velocity.add(0, Player.GRAVITY_STRENGTH);
        if(this.velocity.x>0){
            if(this.velocity.x-Player.AIR_RESISTENCE<0){
                this.velocity.x = 0;
            }else{
                this.velocity.add(-Player.AIR_RESISTENCE,0);
            }
        }else if(this.velocity.x<0){
            if(this.velocity.x+Player.AIR_RESISTENCE>0){
                this.velocity.x = 0;
            }else{
                this.velocity.add(Player.AIR_RESISTENCE,0);
            }
        }
    }

    if(this.velocity.x>Player.MAX_VELOCITY){
        this.velocity.x = Player.MAX_VELOCITY;
    }else if(this.velocity.x<-Player.MAX_VELOCITY){
        this.velocity.x = -Player.MAX_VELOCITY;
    }

    if(this.velocity.y>Player.MAX_VELOCITY){
        this.velocity.y = Player.MAX_VELOCITY;
    }else if(this.velocity.y < -Player.MAX_VELOCITY){
        this.velocity.y = -Player.MAX_VELOCITY;
    }

    this.x += this.velocity.x;
    this.y += this.velocity.y;
}

hit(box){
    if(this.overlap(box)){
        console.log(this, " hit:",box);

        const dist = this.fix(box);
        this.add(dist.x, dist.y);

        if(dist.x!=0){
            this.velocity.x = 0;
        }

        if(dist.y!=0){
            this.velocity.y = 0;
            this.velocity.x = 0;
        }
    }
}

```



```

        }

        if(dist.y < 0){
            this.falling = false;
        }
    }
}

goToPoint(p){
    this.x = p.x;
    this.y = p.y;
    this.velocity.x = 0;
    this.velocity.y = 0;
    this.falling = true;
    this.lastAngle = 0;
}

}

const player = new Player(0, 0, 60, 120);

```

Класът *SavePoint* е отговорен за съхранение на координатите на знамената, запазващи напредъка на играча, както и тяхната визуализация. Знамената, подобно на монетите, са анимирани обекти. За разлика от тях анимацията не се стартира докато героя не влезе в контакт с обекта, а при приключване не се рестартира. Тази проверка се осъществява във функцията *main* и при откриване на контакт се променя стойността на атрибута *active* на *true*.

Методът *draw* отговаря за визуализацията на обекта. По аналогичен начин се изчислява отместването. Изчисляват се и координатите на точките дефиниращи наклонено на 45° знаме. Ако анимацията е активна се извършва ротация на точките спрямо основата с ъгъл, зависещ от номера на текущия кадър от анимацията. Анимацията представлява изправяне на знамето.

```

class SavePoint extends Box{
    static LAST_FRAME = 80;
    constructor(x,y){
        super(x,y, 400, 400);
        this.active = false;
        this.frame = SavePoint.LAST_FRAME;
    }

    draw(ctx, canvas, relativeTo){
        const offsetX = canvas.width/2 - relativeTo.x+this.x;
        const offsetY = canvas.height/2 - relativeTo.y+this.y;

        ctx.beginPath();
        ctx.lineWidth = 8;
        ctx.fillStyle = "rgb(200,200,200)";
        ctx.strokeStyle = "rgb(0,0,0)";

        ctx.rect(0.1*this.width+offsetX, 0.95*this.height+offsetY, this.width*0.15,
this.height*0.05);
        const A = new Vector(0.15*this.width+offsetX, 0.95*this.height+offsetY);
        let B = new Vector(0.15*this.width+offsetX, 0.4*this.height+offsetY);
        let C = new Vector(0.55*this.width+offsetX, 0.4*this.height+offsetY);
        let D = new Vector(0.55*this.width+offsetX, 0.6*this.height+offsetY);
    }
}

```

```

let E = new Vector(0.15*this.width+offsetX, 0.6*this.height+offsetY);

B = rotatePoint(B, A, Math.PI/4);
C = rotatePoint(C, A, Math.PI/4);
D = rotatePoint(D, A, Math.PI/4);
E = rotatePoint(E, A, Math.PI/4);

if(this.active){
    this.frame -= 1;
    if(this.frame<0){
        this.frame = 0;
    }
    B = rotatePoint(B, A, -(1 -
(this.frame/SavePoint.LAST_FRAME))*Math.PI/4);
    C = rotatePoint(C, A, -(1 -
(this.frame/SavePoint.LAST_FRAME))*Math.PI/4);
    D = rotatePoint(D, A, -(1 -
(this.frame/SavePoint.LAST_FRAME))*Math.PI/4);
    E = rotatePoint(E, A, -(1 -
(this.frame/SavePoint.LAST_FRAME))*Math.PI/4);

}

ctx.moveTo(A.x, A.y);
ctx.lineTo(B.x, B.y);
ctx.closePath();
ctx.stroke();
ctx.fill();
ctx.beginPath();
ctx.lineWidth = 1;
ctx.fillStyle = "rgb(230,20,20)";
ctx.moveTo(B.x, B.y);
ctx.lineTo(C.x, C.y);
ctx.lineTo(D.x, D.y);
ctx.lineTo(E.x, E.y);
ctx.closePath()

ctx.stroke();
ctx.fill();
}
}

```

Функцията *main* е отговорна за извикване на методите на свързани с игровата логика(напр. събиране на монети), с графиката (изобразяване на обектите) и за следене дали играта е спечелена или изгубена. При спечелена игра се прекратяват бъдещите извиквания на функцията чрез *clearInterval(startGame)*. *startGame* е променливата съхраняваща таймера извикващ *main* функцията. Таймера се стартира чрез събитието *onload* само след зареждането на страницата. Тя съдържа всички изображения, които се използват от скриптовете под формата на ** тагове. Те не се изобразяват поради стила *style="display: none;"*. По този начин се гарантира, че всички изображения които ще се използват в играта ще са вече заредени.

```

let startGame;
function main(){

```

```

//game code
if(isMouseDown && !player.falling){
    player.jump += Player.JUMP_INCREASE;
    if(player.jump > 1){
        player.jump = 1;
    }
}

player.simulate();

walls.forEach(i =>{
    player.hit(i);
});

let collectedCoins = [];
coins.forEach(i => {
    if(i.overlap(player)){
        collectedCoins.push(i);
        player.score += 1;
    }
});

collectedCoins.forEach(i => {
    const ind = coins.indexOf(i);
    if (ind != -1) {
        coins.splice(ind, 1);
    }
});

savePoints.forEach(i => {
    if(i.overlap(player)){
        player.lastSavePoint = new Vector(i.x+i.width/2, i.y);
        i.active = true;
    }
});

saws.forEach(i => {
    if(i.overlap(player)){
        player.score -= Saw.PENALTY;
        if(player.score < 0){
            player.score = 0;
        }
        player.goToPoint(player.lastSavePoint);
        screenBlur.begin();
    }
});

let gameOver = false;
if(endStar.overlap(player)){
    setWin(canvas);
    gameOver = true;
}

//////////graphics code//////////
let drawCount = 0;
//update canvas dimensions to match the window
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;

//center the screen on this point
let focusPoint = new Vector(player.x+player.width/2, player.y+player.height/2);

```

```

screenBlur.draw(ctx);

clear(ctx, canvas, focusPoint);

screen.x = focusPoint.x - canvas.width/2;
screen.y = focusPoint.y - canvas.height/2;
screen.width = canvas.width;
screen.height = canvas.height;

//draw decorations
decorations.forEach(i => {
  if(i.overlap(screen)){
    drawCount += 1;
    i.draw(ctx, canvas, focusPoint);
  }
});

//draw coins
coins.forEach(i => {
  if(i.overlap(screen)){
    drawCount += 1;
    i.draw(ctx, canvas, focusPoint);
  }
});

//draw the player
drawCount += 1;
player.draw(ctx, canvas, focusPoint);

//draw walls
walls.forEach(i =>{
  if(i.overlap(screen)){
    drawCount += 1;
    i.draw(ctx, canvas, focusPoint);
  }
});

//draw saws
...

//draw save points
...

//draw star
if(endStar.overlap(screen)){
  drawCount += 1;
  endStar.draw(ctx, canvas, focusPoint);
}

if(isMouseDown && !player.falling){
  drawJumpBar(ctx, canvas, mousePos, player.jump);
}

drawScore(player.score);
console.log("drawing: ", drawCount, " objects");

if(gameOver){
  score.innerHTML = "You win!\nScore:"+player.score;
  clearInterval(startGame);
}

```

```

    }
}

//for testing
document.addEventListener("keydown", event => {
    if(event.key == 'g'){
        let newX = parseInt(prompt("x:"));
        let newY = parseInt(prompt("y:"));
        player.goToPoint(new Vector(newX, newY));
    }
});

window.onload = function() {
    startGame = setInterval(main, 1000/60);
}

```

Файлът „input.js“ съдържа функции и променливи, отговорни за получаване на входните данни от играча (управлението от мишката). Променливата *mousePos* съхранява текущото местоположение на курсора, а *isMouseDown* – булева стойност, показваща дали е задържан левият бутон на мишката. Тези променливи се актуализират съответно от функциите *mouseMove* и *mouseDown/mouseUp*, които се извикват при настъпване на съответните събития от *EventListener*. Функцията *mouseUp* е допълнително отговорна за извършване на подскоци. Тя създава вектор с начало – центъра на екрана и край – местоположението на мишката. Вектора се преобразува, така че дължината му да е пропорционално равна на силата на подскока. Той се прибавя към вектора на скоростта на героя, който е (0,0) тъй като подскок може да се извърши само ако героя не се движи.

```

let mousePos = new Vector(0,0);
let isMouseDown = false;

function mouseMove(event){
    mousePos.x = event.clientX;
    mousePos.y = event.clientY;
    //console.log("mouse move:", mousePos);
}

function mouseDown(){
    isMouseDown = true;
}

function mouseUp(){
    isMouseDown = false;
    if(!player.falling){
        const direction = new Vector(mousePos.x -
document.getElementById('canvas').width/2 , mousePos.y -
document.getElementById('canvas').height/2);
        console.log("jump:", direction);
        normaliseVector(direction, Player.MAX_JUMP*player.jump);
        player.falling = true;
        player.velocity.add(direction.x, direction.y);
        player.jump = 0;
    }
}

```

```

}

document.getElementById('canvas').addEventListener('mousemove', (event) =>
mouseMove(event));
document.getElementById('canvas').addEventListener('mousedown', (event) =>
mouseDown());
document.getElementById('canvas').addEventListener('mouseup', (event) =>
mouseUp());

```

Функцията *clear*, съдържаща се във файла *clear.js*, рисува градиент, който представлява фона на сцената. По този начин също се изчиства изцяло предходния кадър. Цветовете на градиента зависят от местоположението на героя, което се подава като параметър на функцията. Функцията се извиква преди изрисоването на всеки кадър.

```

function clear(ctx, canvas, location){
    let gradient = ctx.createLinearGradient(0, 0, 0, canvas.height);

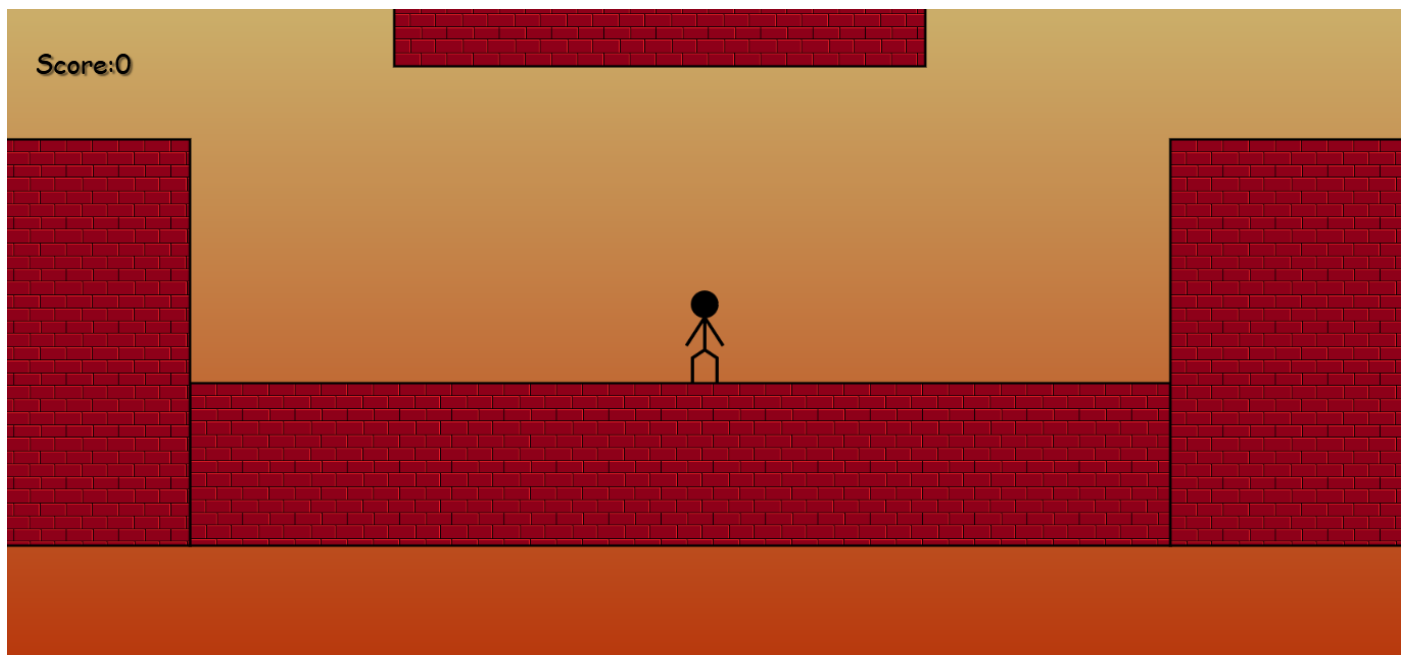
    if(location.y > -1900){
        gradient.addColorStop(0, "rgb(200, 170, 100)");
        gradient.addColorStop(1, "rgb(180,50,10)");
    }else if(location.y > -4800){
        gradient.addColorStop(0, "rgb(100, 100, 110)");
        gradient.addColorStop(1, "rgb(50,50,55)");
    }else{
        gradient.addColorStop(0, "rgb(10, 120, 50)");
        gradient.addColorStop(1, "rgb(0,60,20)");
    }
    ctx.fillStyle = gradient;
    ctx.fillRect(0, 0, canvas.width, canvas.height);
}

```

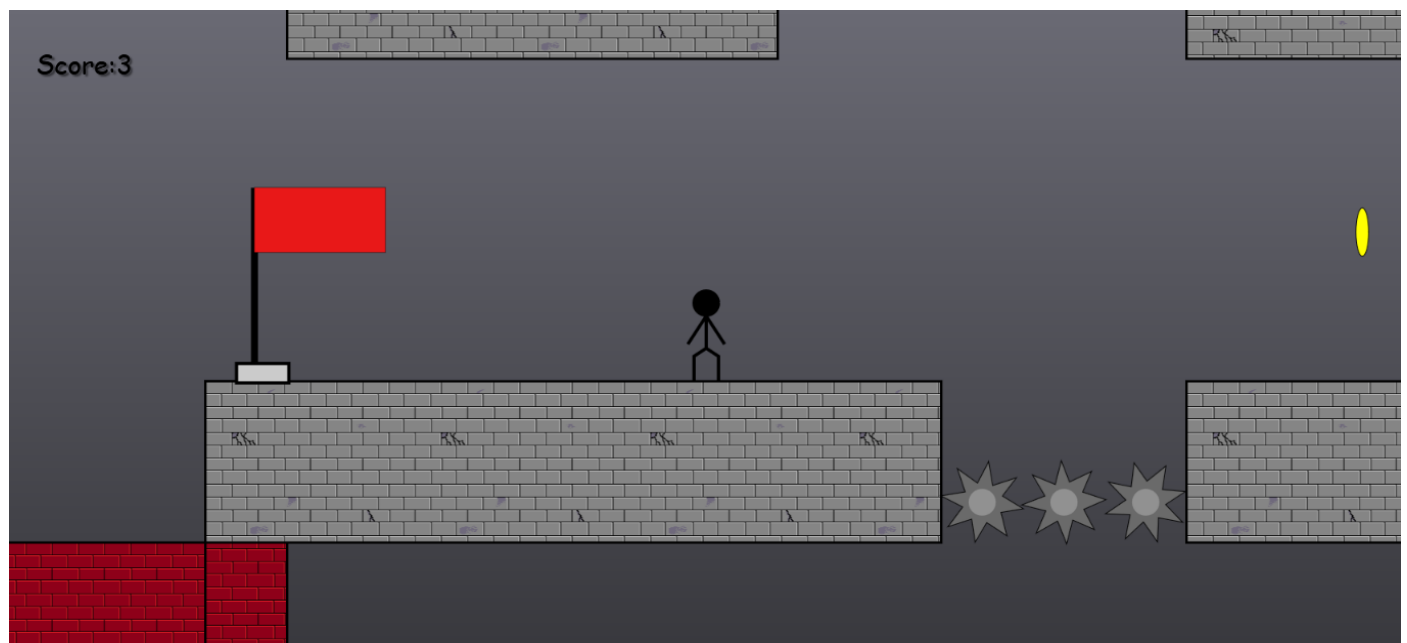
5. Демонстрационни кадри

В следния раздел са представени кадри от работата играта.

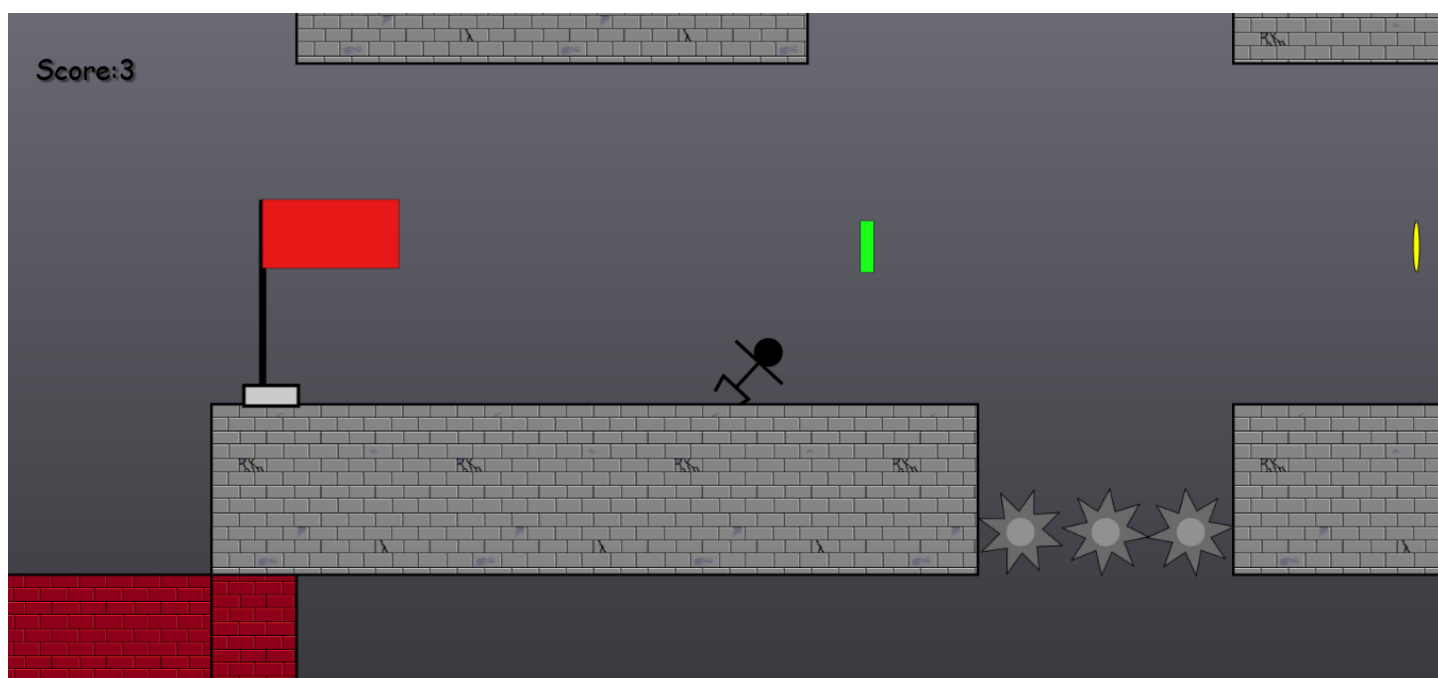
Начално състояние на екрана



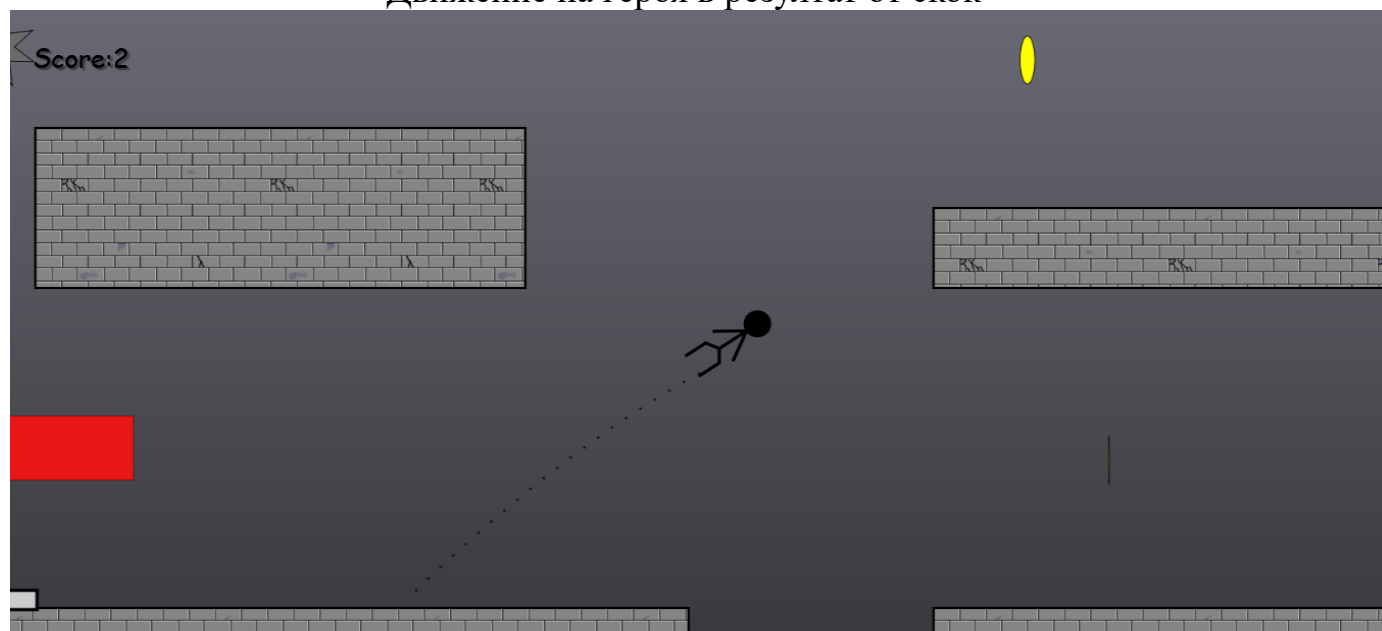
Втора част от нивото



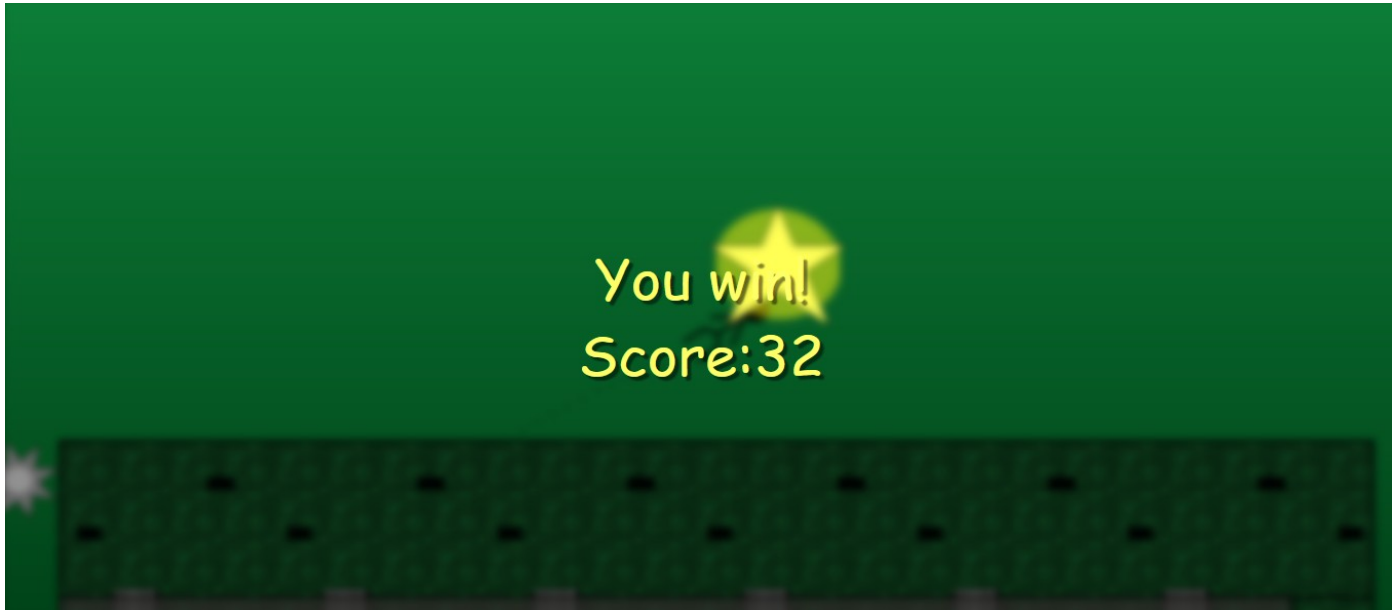
Подготовка за извършване на скок от героя



Движение на героя в резултат от скок



Финален екран на играта



6.Заклучение

Проектът използва различни техники и алгоритми от компютърната графика, дизайн на игрите както и библиотечни функции на езика Javascript. За бъдещо подобрене на проекта може да се добавят допълнителни нива, движещи се препятствия и платформи както и допълнително декоративни обекти.

Проектът е достъпен в Github: <https://github.com/IvanDimovSIT/GS-Project>

използвани източници:

https://eloquentjavascript.net/17_canvas.html

<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>

<https://developer.mozilla.org/en-US/docs/Web/CSS/gradient/linear-gradient>

<https://stackoverflow.com/questions/20253210/canvas-pattern-offset>

<https://stackoverflow.com/questions/1795100/how-to-exit-from-setinterval>

<https://stackoverflow.com/questions/15255801/javascript-addeventlistener-function>