Lesson 03 - Theoretical Considerations and Code Optimization

Now it's getting a little more complex. Our game still lacks a collision query, which allows us to bounce the ball off the paddles and the edges of the screen. In this lesson, we prepare to program such a query. We will also expand our game with walls that represent the edge of the field and at the same time optimize the code. After all, according to good software design, a function should never have more than 20 lines of code.

But first to the planning of the collision query.
We should focus on two questions:

1. When does a collision take place??

To put it simply, we can say that a collision always occurs when the centre of the ball lies within the rectangle of another object.

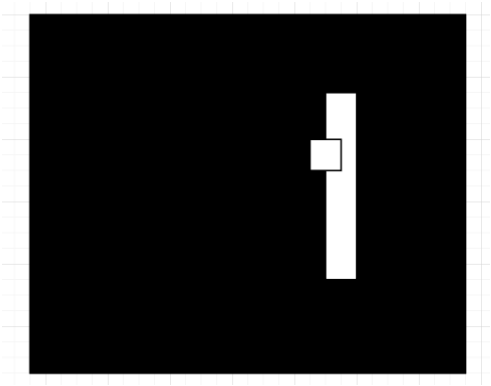

Figure 1: The centre of the ball is on the edge of a paddle

2. What should happen if a collision is detected?

The Movement of the ball should be reflected and not just inverted so that the ball changes direction instead of just flying back.

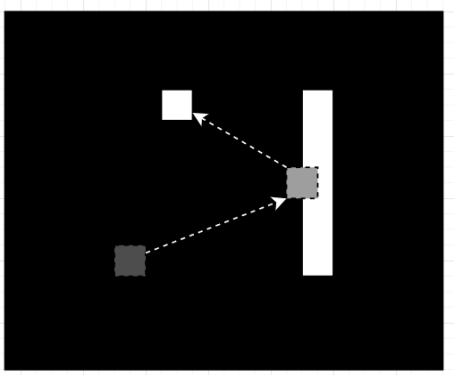

Figure 2: The paddle reflects the ball

The answers give us the basis for a flow chart, which in turn will help us with the programming.

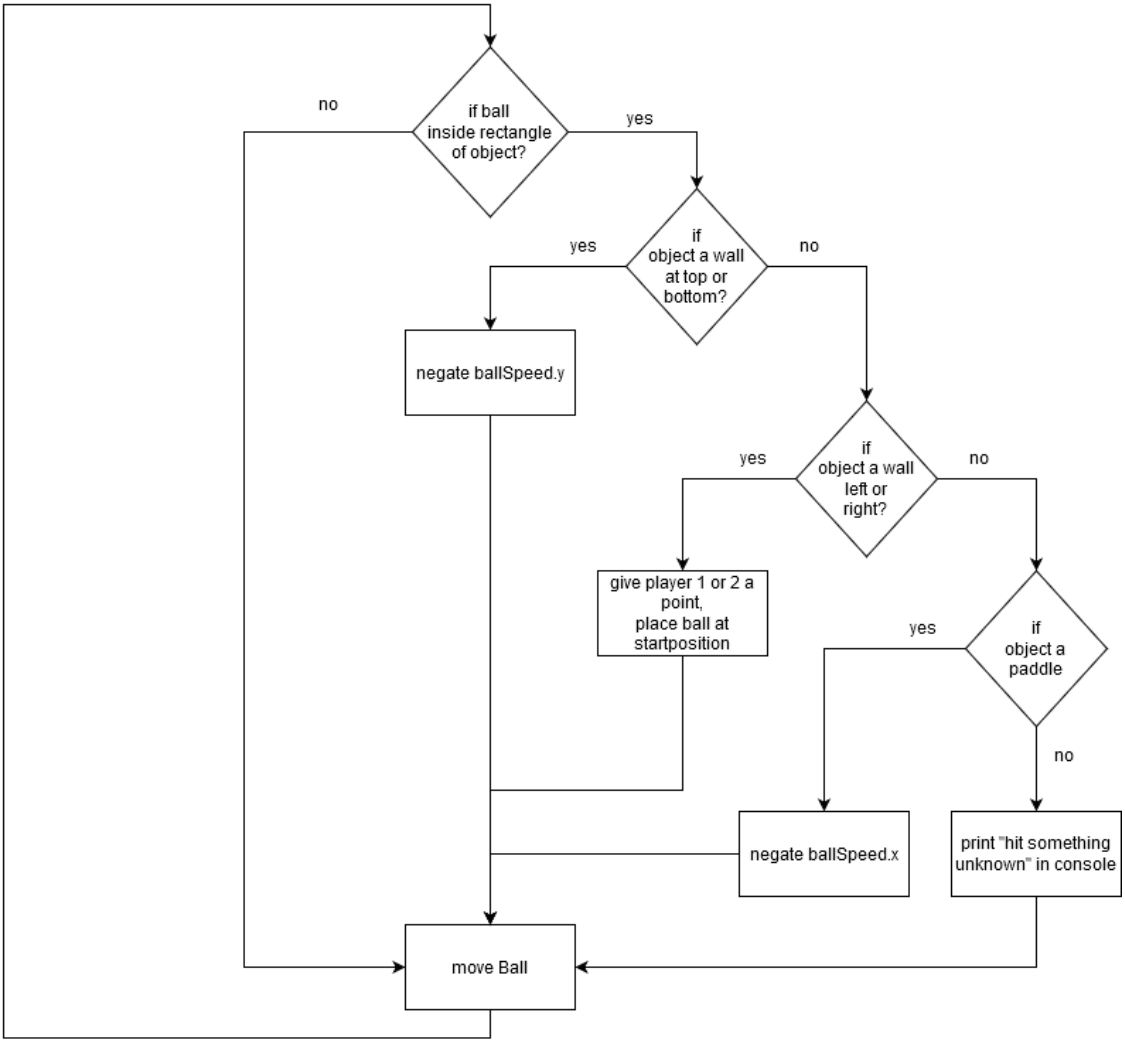

Figure 3: Flow chart for collision detection

Now for inserting the walls and code optimization.

For the walls we need four more nodes, which are all part of the level. Our scene tree from lesson 01 expands by four branches.

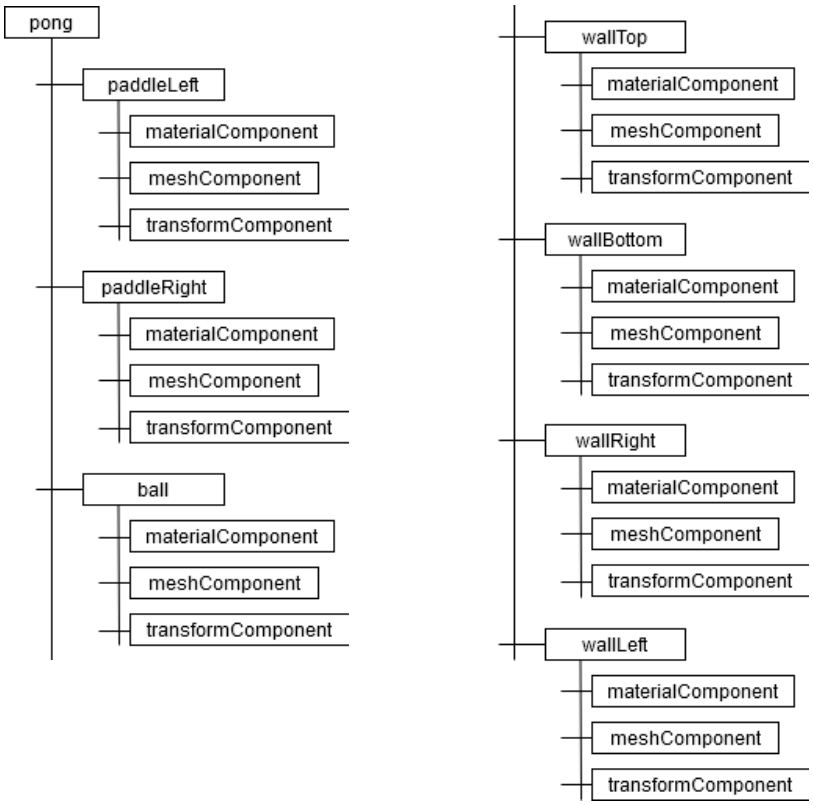

Figure 4: Extended scene tree

Since it would unnecessarily inflate the code if we created these four nodes in the same way as those of the ball and the paddles, it is advisable to outsource the creation of a node to a *createNode function*.

createNode takes five values (name, mesh, material, position and size) and returns a node. The five values are used to create a complete node with all the necessary components. *createNode* is then called in *createPong* while the four walls are attached to the pong node.

```
function createNode(_name: string, _mesh: ƒ.Mesh,_material: ƒ.Material,
  _translation: ƒ.Vector2, _scaling: ƒ.Vector2): ƒ.Node {
    let node: ƒ.Node = new ƒ.Node(_name);
    node.addComponent(new ƒ.ComponentTransform);
    node.addComponent(new ƒ.ComponentMaterial(_material));
    node.addComponent(new ƒ.ComponentMesh(_mesh));
    node.cmpTransform.local.translate(_translation.toVector3());
```

```
        node.getComponent(ƒ.ComponentMesh).pivot.scale
            (_scaling.toVector3());

        return node;
    }
```

```
    function createPong(): ƒ.Node {
        let pong: ƒ.Node = new ƒ.Node("Pong");

        let mtrSolidWhite: ƒ.Material = new ƒ.Material("SolidWhite",
            ƒ.ShaderUniColor, new ƒ.CoatColored(ƒ.Color.CSS("WHITE")));
        let meshQuad: ƒ.MeshQuad = new ƒ.MeshQuad();

        pong.appendChild(createNode("WallLeft", meshQuad, mtrSolidWhite,
            new ƒ.Vector2(-22, 0), new ƒ.Vector2(1, 30)));
        pong.appendChild(createNode("WallRight", meshQuad, mtrSolidWhite,
            new ƒ.Vector2(22, 0), new ƒ.Vector2(1, 30)));
        pong.appendChild(createNode("WallTop", meshQuad, mtrSolidWhite,
            new ƒ.Vector2(0, 15), new ƒ.Vector2(45, 1)));
        pong.appendChild(createNode("WallBottom", meshQuad, mtrSolidWhite,
            new ƒ.Vector2(0, -15), new ƒ.Vector2(45, 1)));

        ball = createNode("Ball", meshQuad, mtrSolidWhite,
            ƒ.Vector2.ZERO(), new ƒ.Vector2(1, 1));
        paddleLeft = createNode("PaddleLeft", meshQuad, mtrSolidWhite,
            new ƒ.Vector2(-20, 0), new ƒ.Vector2(1, 4));
        paddleRight = createNode("PaddleRight", meshQuad, mtrSolidWhite,
            new ƒ.Vector2(20, 0), new ƒ.Vector2(1, 4));

        pong.appendChild(ball);
        pong.appendChild(paddleLeft);
        pong.appendChild(paddleRight);

        return pong;
    }
```

In the sample code, the walls were placed so that they are outside the screen window. If you want, you can also place them visibly or experiment with the colours of the walls.

Next, we can optimize the controls. We can also outsource the four if statements for the controls in the *update function*.
In addition, an associative array *controls* can be created for the controls and an interface *control* can be used to transfer to the array how and what is controlled when a certain key is pressed. The two values in the interface are *paddle* of type Node (which paddle is controlled) and *translation* of type Vector3 (which movement should be carried out). In a function *defineControls*, which returns *controls*, we can assign each key what it should do and save this in the associative array.
With the *processInput function*, we can use a for-loop to go through all entries in the array *controls* and control the appropriate paddle based on the values in the interface *control*.

```
namespace Pong {
    import ƒ = FudgeCore;

    interface KeyPressed {
        [code: string]: boolean;
    }

    interface Control {
        paddle: ƒ.Node;
        translation: ƒ.Vector3;
    }

    interface Controls {
        [code: string]: Control;
    }

    let viewport: ƒ.Viewport;

    let pong: ƒ.Node;
```

```
let ball: ƒ.Node;
let mtxBall: ƒ.Matrix4x4;
let paddleLeft: ƒ.Node;
let paddleRight: ƒ.Node;
let keysPressed: KeyPressed = {};
let ballSpeed: ƒ.Vector3 = new ƒ.Vector3(0.5, -0.0, 0);
let paddleSpeedTranslation: number = 0.5;
let paddleSpeedRotation: number = 5;
let controls: Controls;

window.addEventListener("load", hndLoad);

function hndLoad(_event: Event): void {
    const canvas: HTMLCanvasElement = document.querySelector("canvas");
    ƒ.RenderManager.initialize();


    pong = createPong();
    controls = defineControls();
    mtxBall = ball.cmpTransform.local;

    let cmpCamera: ƒ.ComponentCamera = new ƒ.ComponentCamera();
    cmpCamera.pivot.translateZ(42);
    cmpCamera.pivot.lookAt(ƒ.Vector3.ZERO());
    viewport = new ƒ.Viewport();
    viewport.initialize("Viewport", pong, cmpCamera, canvas);

    document.addEventListener("keydown", hndKeydown);
    document.addEventListener("keyup", hndKeyup);

    viewport.draw();

    ƒ.Loop.addEventListener(ƒ.EVENT.LOOP_FRAME, update);
    ƒ.Loop.start();
}
```

```
    function update(_event: Event): void {
        processInput();

        moveBall();
        viewport.draw();
    }

    function processInput(): void {
        for (let code in controls) {
            if (keysPressed[code]) {
                let control: Control = controls[code];
                let mtxPaddle: ƒ.Matrix4x4 =
                  control.paddle.cmpTransform.local;

                  mtxPaddle.translation =
                        ƒ.Vector3.SUM(mtxPaddle.translation,
                        control.translation);
            }
        }
    }

    function defineControls(): Controls {
        let controls: Controls = {};
        controls[ƒ.KEYBOARD_CODE.ARROW_UP] = { paddle: paddleRight,
            translation: ƒ.Vector3.Y(paddleSpeedTranslation)};

        controls[ƒ.KEYBOARD_CODE.ARROW_DOWN] = { paddle: paddleRight,
            translation: ƒ.Vector3.Y(-paddleSpeedTranslation)};

        controls[ƒ.KEYBOARD_CODE.W] = { paddle: paddleLeft,
            translation: ƒ.Vector3.Y(paddleSpeedTranslation)};

        controls[ƒ.KEYBOARD_CODE.S] = { paddle: paddleLeft,
            translation: ƒ.Vector3.Y(-paddleSpeedTranslation) };

        return controls;
    }
```

```
function moveBall(): void {
    mtxBall.translate(ballSpeed);
}

function hndKeyup(_event: KeyboardEvent): void {
    keysPressed[_event.code] = false;
}
function hndKeydown(_event: KeyboardEvent): void {
    keysPressed[_event.code] = true;
}

function createPong(): ƒ.Node {
    let pong: ƒ.Node = new ƒ.Node("Pong");

    let mtrSolidWhite: ƒ.Material = new ƒ.Material("SolidWhite",
        ƒ.ShaderUniColor, new ƒ.CoatColored(ƒ.Color.CSS("WHITE")));
    let meshQuad: ƒ.MeshQuad = new ƒ.MeshQuad();

    pong.appendChild(createNode("WallLeft", meshQuad, mtrSolidWhite,
        new ƒ.Vector2(-22, 0), new ƒ.Vector2(1, 30)));
    pong.appendChild(createNode("WallRight", meshQuad, mtrSolidWhite,
        new ƒ.Vector2(22, 0), new ƒ.Vector2(1, 30)));
    pong.appendChild(createNode("WallTop", meshQuad, mtrSolidWhite,
        new ƒ.Vector2(0, 15), new ƒ.Vector2(45, 1)));
    pong.appendChild(createNode("WallBottom", meshQuad, mtrSolidWhite,
        new ƒ.Vector2(0, -15), new ƒ.Vector2(45, 1)));

    ball = createNode("Ball", meshQuad, mtrSolidWhite,
        ƒ.Vector2.ZERO(), new ƒ.Vector2(1, 1));
    paddleLeft = createNode("PaddleLeft", meshQuad, mtrSolidWhite,
        new ƒ.Vector2(-20, 0), new ƒ.Vector2(1, 4));
    paddleRight = createNode("PaddleRight", meshQuad, mtrSolidWhite,
        new ƒ.Vector2(20, 0), new ƒ.Vector2(1, 4));

    pong.appendChild(ball);
    pong.appendChild(paddleLeft);
```

```
        pong.appendChild(paddleRight);

        return pong;
    }

    function createNode(_name: string, _mesh: ƒ.Mesh,_material: ƒ.Material,
      _translation: ƒ.Vector2, _scaling: ƒ.Vector2): ƒ.Node {
        let node: ƒ.Node = new ƒ.Node(_name);
        node.addComponent(new ƒ.ComponentTransform);
        node.addComponent(new ƒ.ComponentMaterial(_material));
        node.addComponent(new ƒ.ComponentMesh(_mesh));
        node.cmpTransform.local.translate(_translation.toVector3());
        node.getComponent(ƒ.ComponentMesh).pivot.scale
            (_scaling.toVector3());

        return node;
    }
}
```

In the next lesson, we'll put our considerations of the collision into practice.