MINISTERUL EDUCAŢIEI REPUBLICII MOLDOVA

UNIVERSITATEA TEHNICĂ A MOLDOVEI

Facultatea „Calculatoare, Informatică şi Microelectronică”

FILIERA ANGLOFONĂ

RAPORT

Lucrare de laborator #1

la APPOO

A efectuat:

st. gr. FAF-151 Bîzdîga Stanislav

A verificat:

asist.univ. Șerșun Anastasia

Chișinău-2018

Project Description

Pong is one of the earliest arcade video games. It is a table tennis sports game featuring simple two-dimensional graphics.[1]

Inspiration: Pong (Atari 1972)

Game Features:
- Two player game.
- Dual user controlled paddles: Left side / Right side.
- Simple controls: “up” & “down” / “w” & “s”.
- Secret: Advanced controls: “left” & “right” / “a” & “d” - for defying the angle.
- A constant-velocity bouncy ball **(with on-hit acceleration)**.
- Ball infinite ricochet avoidance system. (adds angle when going straight)
- Ball reset/respawn in center at round loss.
- Random direction launch of the respawned ball.
- Predefined 8 possible paths of ball launch. (avoiding infinite ricochet loops)
- A score system that triggers the win of the respective player due to reaching 10 points.
- A fancy title screen and win screen with animations and particle effects.
- 4 in-game pumping sound tracks (randomly picked)
- 3 win-screen cool songs (randomly picked)
- 11 fun and engaging sound effects.

Game look in a nutshell:

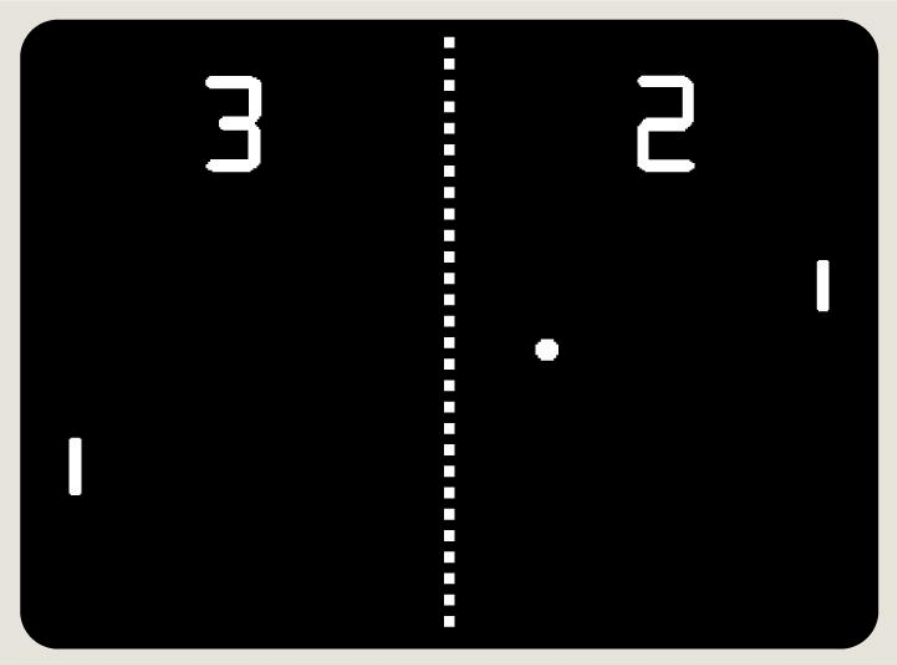

fig.1 Slightly modern looking Pong

Task realization

Object-oriented programming is a programming paradigm relying on the idea of "objects", that can keep in some data, called attributes; and functions, called methods. OOP is a helping hand regarding modelling a system or problem in a more readable, and understandable manner, using the form described above. The main principles of OOP with included examples are coming up ahead:

The main principles of OOP:

1.Encapsulation
Encapsulation is the idea that the attributes of an entity are enclosed in that entity. This gives context to attributes. This also allows the programmer to restrict access to those attributes so that those attributes are modified and/or used only in ways that the programmer intends to use them.
Code example:

```
public class GameController : MonoBehaviour {
    private static bool isGameOn;
    private static bool isGameOver;
...

}
```

Thus, by having a game controller class, the inner attributes like isGameOver or isGameOn are logically connected to the object of the respective class, making the modelling of the problem easier and more obvious.
Another example, regarding the ball behaviour:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour {
    public float constantSpeed = 10f;
    private Rigidbody rb;

    void Start () {
        // Reset init params
        constantSpeed = 10f;

        // storing the rigidbody in a var for code cleanliness
        rb = this.GetComponent<Rigidbody> ();

        // Pause game before launching the ball
        GameController.instance.PauseGame ();
```

```
        // Granting random orientation(angle) to the ball launch
        rb.velocity =
            new Vector3(2f * Random.Range(1,3) - 3f,          // -1 or 1 on X
                        2f * 0.9f * Random.Range(1,3) - 3f,
                        0f );
    }

    void LateUpdate () {

        // every frame set the speed of the ball to remain constant
        rb.velocity = constantSpeed * (rb.velocity.normalized);

        // prevention of straightly horizontal, endless looping pongs
        if (rb.velocity.y == 0f)
        {
            // adding some velocity on y axis
            rb.velocity =
                new Vector3(rb.velocity.x,
                    2f * 0.9f * Random.Range(1,3) - 3f, //here
                    0f );
        }
    }

    private void addSpeed()
    {
        constantSpeed += 0.25f;
    }

    void OnTriggerEnter (Collider other)
    {
        ScreenShake.Shake (0.1f,0.005f, rb.velocity.x, rb.velocity.y);
        addSpeed ();
    }
}
```

Let's take the float variable containing the constant speed, for example. As it is clearly visible, the attribute of the ball's constant speed is encapsulated in the current class, being private, thus having no outer accessibility, which is due to the fact that it's only needed and handled within the current class.

Obviously, this behaviour is attached to the ball object in the scene and it has some specific procedures that are running during the game. These include: OnTriggerEnter - a function called when the object enters a trigger box (or touches it), addSpeed - a custom function, that adds a bit of speed to the ball, and the Start method, which is run once, when the object is instantiated.
There is also the LateUpdate method which is called every frame.

2.Inheritance

Inheritance is the idea that an entity can inherit attributes and methods from another entity. It allows the programmer to create similar entities without needing to rewrite similar code over and over.

Note that using the unity engine, the script is automatically inheriting from a MonoBehaviour object, like:

```
public class BallBehaviour : MonoBehaviour{
...
}
```

MonoBehaviour is the base class from which every Unity script derives. It contains these basic functions:

Start()
Update()
FixedUpdate()
LateUpdate()
OnGUI()
OnDisable()
OnEnable()

Inheritance allows the use of these functions and accessing of parameters from the superior object.

3.Polymorphism

Polymorphism means “having many forms”. It can be used in two ways:

Overloading: The method name stays the same, but the parameters, the return type and the number of parameters can all change.

Code exapmle:

```
    public void PlayOtherSfx(params AudioClip[] clips) {
        int randomIndex = Random.Range(0, clips.Length);
        //Set the clip to the clip at our randomly chosen index.
        spcSource.clip = clips[randomIndex];
        //Set the pitch of the audio source to the randomly chosen pitch.
        spcSource.pitch = Random.Range(lowPitchRange, highPitchRange);
        //Play the clip.
        spcSource.Play ();
    }
```

```
    //overloading PlayOtherSfx to play w/ specific audiosource and audioclip
    public void PlayOtherSfx(AudioSource a, AudioClip c)
    {
        a.clip = c;  // insert audioclip in audiosource
        a.Play();    // play it
    }
```

Overriding: This is when a derived class method has the same name, parameters and return type as a method in a base class but has a different implementation.

Code exapmle: Player.cs

```
using UnityEngine;
using System;
using System.Collections;

...
namespace PlayerObject{
public abstract class PlayerDefault{

    protected float moveSpeed = 8f;
    protected GameObject player;
    public abstract void allowMovement();
}

public class Player1 : PlayerDefault {

    private float moveSpeed;

    public Player1(){
        player = GameObject.FindGameObjectWithTag ("Player1");
    }

    public override void allowMovement(){
      ...
      // Player controls implemented with WASD here
      ...
}
}

public class Player2 : PlayerDefault {

    private float moveSpeed;

    public Player2(){
        player = GameObject.FindGameObjectWithTag ("Player2");
    }

    public override void allowMovement(){
      ...
      // Player controls implemented with arrow keys here
      ...
    }
}
} // enclose namespace
```

Deriving from the comments, the implementation of controls are different but the name of the function is completely similar, and the calls would look like:

```
using System.Collections;
using UnityEngine;
using PlayerObject;

public class UserController : MonoBehaviour {
    ...
    Player1 player1;
    Player2 player2;

    void Start () {
        player1 = new Player1();
        player2 = new Player2();
        }

    void Update () {
           ...
            player1.allowMovement();
            player2.allowMovement();
           ...
     }
...
}
```

Thus, it is the overriding property of polymorphism. And lastly:

4.Abstraction

Abstraction is the process of hiding all but the relevant information about a thing to make things less complex and more efficient for the user. For example, we don't need to know how a fridge works in order to use its conserving benefits. Abstraction lets one focus on what the thing does instead of how it does it. **Abstract classes**: This is an empty class, with only the containers of the functions (method declarations known as prototypes). It doesn't specify how is the method implemented but only the fact that it exists. The derived classes MUST implement the abstract methods, and abstract classes can not be instantiated.

See previous code example for an example of abstract class as well, since the PlayerDefault class is not usable, it is just a container for the derived classes Player1 and Player2 which have different user-input controls implementations.

Interfaces: They are alike to abstract classes, however, a multitude of interfaces are able to be implemented(inherited) by a single class. It expands the ability of making a good use of abstraction. Classes that implement interfaces must implement interface's methods, otherwise it throws an error.

```
using UnityEngine;
using System;
using System.Collections;

namespace PlayerObject{

    public interface Movement{
        void moveY();
    }

    public interface Rotation{
        void turnZ();
    }

public abstract class PlayerDefault{

    protected GameObject player;
    protected float moveSpeed = 8f;     // paddle speed
    protected float yLimit = 3.5f;      // y axis boundary (amplitude)

    public abstract void allowMovement ();
}

public class Player1 : PlayerDefault, Movement, Rotation {

    public Player1(){
            player = GameObject.FindGameObjectWithTag ("Player1");
     // alternatively could instantiate from prefab
    }

    public override void allowMovement(){ //(<-left player)
        // P1 Move controls
        moveY();
        // P1 Rotation controls
        turnZ();
    }

    public void moveY(){  // function implementation from interface
    ...     }
    public void turnZ(){  // function implementation from interface
    ...     }
}
```

```
public class Player2 : PlayerDefault, Movement, Rotation {
        public Player2(){
            player = GameObject.FindGameObjectWithTag ("Player2");
        }

    public override void allowMovement(){ //(right player->)
        // P2 Move controls
        moveY();
        // P2 Rotation controls
        turnZ();
    }

    public void moveY(){ // other function implementation from interface
    ...     }

    public void turnZ(){ // other function implementation from interface
    ...     }
}


}
```

Testing the results

Game looks:

fig.2 Title screen

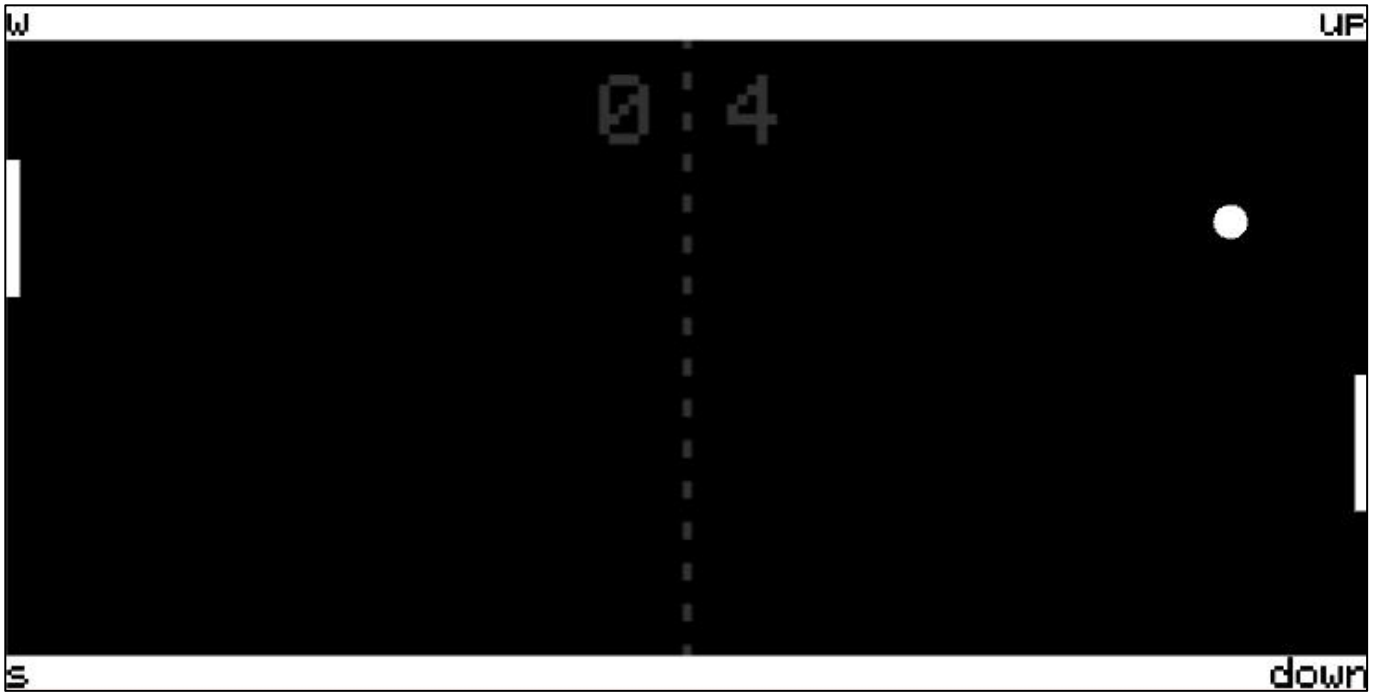

fig.3 In-game screen

fig.4 Win screen (Text and background are animated)

There is some replayability to this game, since you have different music each time you play.
(It is randomly picked among the provided resources.)

Also, the ball is thrown arbitrarily in different directions at random, so that luck can play a role in it as well. It is a fun, challenging game that was done in a very short time. Fair enough, it felt impressive like a Ludum Dare.

Regarding the bugs:
There has been some bugs during production that quickly and swiftly have been removed and fixed. So as far the solo playtesting went, it did not reveal any bugs whatsoever. If you want to try it for yourself, it is available for download in the following archive:

Pong-EarlyAccess.rar at:
https://github.com/StasBizdiga/APPOO/tree/master/Lab1/[APPOO]Lab1/BUILD

Conclusion

The current laboratory work was very interesting and challenging.
It was a new, unusual, and thus captivating experience. Having such freedom and trying to set the challenges by self in the way one sees it personally was very entertaining and motivating. Everyday laboratories and tasks are mundane and boring compared to this approach. Students are used to clear requirements, but life is not always precise with what you need to do or choose. Therefore it's a good point.
The main tasks, nevertheless, were specified clearly, and thanks to that, the algorithms and logic behind object oriented programming cleared up. It was very much worth it.

Bibliography

- **[1] Pong**
  https://en.wikipedia.org/wiki/Pong
- **[2] Four Principles of OOP**
  http://scottpantall.com/2017/09/four-principles-object-oriented-programming-examples-c/

Annex

Code source: GitHub
https://github.com/StasBizdiga/APPOO