TP Videojuegos
Exercise 1

In the first few classes of this semester, we have seen how to use composition instead of inheritance in order to define behaviors (input, physics, and rendering) of a `GameObject`. In this exercise you will practice this idea, starting from the Ping-Pong code that we have written in class. Most of the code you develop in this exercise will be reused in assignment 1, in which you will develop a variation of the classic Asteroids game.

The overall goal of this exercise is to implement an actor that represents a space fighter (kind of an aircraft) that can is able to move in the space, rotate, accelerate, decelerate, and shoot.

You will be guided to do this exercise step-by-step, where at each step you will implement a new component and use it to improve the behavior of the actor(s).

Step 1: `ImageRenderer`

Implement a `RenderComponent` that allows drawing a `GameObject` as an image (provided as a `Texture`) taking into account its direction, i.e., rotating the image such that its front matches the direction of the corresponding `GameObject`. Start from the class `ImageRenderer` available in the provided code and modify it with the rotation behavior.

We will assume that images (in the png files) are always facing up, i.e., in the direction of the vector `(0,-1)`. Thus, when drawing the image it should be rotated by α degrees, where α is the angle between the vector `(0,-1)` and the vector `o->getDirection()` of the `GameObject o`. Note that class `Vector2D` has a method to calculate the angle between two vectors. Recall that the y-axis of our coordinate system is inverted to make it easier work with `SDL`.

Note that the class `Texture` has a `render` method that receives the rotation angle as a parameter.

Testing: create a `GameComponent` that represents the fighter, and add to it an instance of `ImageRenderer` using the image texture `Resources::SpaceShips` or `Resources::Airplanes`(pick any of the images from the corresponding `png` by defining an appropriate `clip`). Note that this should be done in the class `ExampleGame.cpp` (declare corresponding fields, and modify methods `initGame()`

and `closeGame()` accordingly). Set the initial direction of the fighter's component to some vector in order see how the image is rotated.

Step 2: **`RotationInputComponent`**

Implement an `InputComponent` that allows rotating a `GameObject`. The constructor should receive the rotation angle α and two keys to be used for the *clockwise* and *counter-clockwise* rotation.

When the clockwise rotation key is pressed the corresponding `GameObject` is rotated by α degrees, and when the counter-clockwise rotation key is pressed it is rotated by $-\alpha$ degrees. Note that class `Vector2D` has a method to rotate a vector by α degrees.

Testing: create an instance of `RotationInputComponent` that is controlled by the left and right arrows, with a rotation angle of 5 degrees, and add it to the fighter's component.

Step 3: **`CircularMotionPhysics`**

Implement a `PhysicsComponent` that allows a `GameObject` to move in a circular way, i.e., when it completely exits from one side it appears in the opposite side. Note that in each call to method `update` the new position of the object is first changed to `o->getPosition()+o->getVelocity()`, and then its `X` and `Y` components are modified accordingly if it exits the borders of the window, etc.

Testing: create an instance of `CircularMotionPhysics` and add it to the fighter's component. Set the default velocity of the fighter's component to `(1,2)`, otherwise, it won't move since the velocity by default is `(0,0)`. Later, when the fighter is provided with the possibility to accelerate, you should put it back to `(0,0)`.

Step 4: **`AccelerationInputComponent`**

Now we will implement an `InputComponent` that allows the fighter to accelerate and decelerate. The component's constructor receives as parameters two keys for acceleration and deceleration.

<u>Acceleration</u>: note that the fighter is facing in the direction of the vector `getDirection()` while it is moving in the direction of the vector `getVelocity()`. Accelerating is not simply increasing the magnitude of the velocity vector, but also modifying the direction of the velocity vector gradually such that at the end the fighter moves in the same direction that it is facing, i.e., the directions of `getDrection()` and

`getVelocity()` are equal. This can be changing the velocity of the object to `getVelocity()+getDirection()*thrust`, where `thrust` is a number (of type `double`) representing some power that is applied on the fighter to change its motion direction. Make the `thrust` parameter of the constructor. The velocity should have a maximum limit that is provided as a parameter to the constructor, when exceeded you should fir normalize the velocity vector (i.e., make its magnitude to be 1 using `v.normalize()`) and then multiply it by the velocity limit.

Deceleration: Choose one of the following option for reducing the velocity, or implement both if you like:

A. Reduce the magnitude of the velocity vector `v` by a constant `c`, which can be done as follows. If `v.magnitude()>=c>0`, then the velocity vector is set to `v*(v.magnitude()-c)/v.magnitude()`, otherwise it is set to `(0,0)`.

B. Reduce the magnitude of the velocity vector `v` by a reduction factor between 0 and 1, i.e., set the velocity vector to `v*reductionFactor`.

Note that `c` and `reductionFactor` should be parameters of the constructor. When the magnitude of the velocity vector is smaller than some epsilon we set it to `(0,0)`.

Testing: create an `AccelerationInputComponent` that is controlled by the `up` and `down` arrows. Try different values for `thrust` (e.g., between 0.5 and 1) to see how it affects the motion of the fighter.

Step 5: `BulletsManager`

Create an interface that represents a bullets manager. The interface should include a single method `shoot(GameObject* o, Vector2D p, Vector2D v)`. This method is supposed to be called to create a bullet with initial position `p` and velocity `v`. Note that it is just an interface, the implementation will come in the next step.

Step 6: `StarWarsBulletsManager`

Implement a `GameObject` that is also a `BulletsManager`, i.e., it inherits from both `GameObject` and `BulletsManager`. It should maintain a list of bullets (e.g., `std::vector<GameObject*>`) which represent the bullets that are currently on the screen. Next we describe methods `shoot`, `update` and `render`.

`shoot`: creates a new `GameComponent` with components `FillRectRenderer` and `BasicMotionPhysics` (both are given), with size small enough, e.g., 2x2, and adds it to the list of bullets. Note that you should use the same instance of classes `FillRectRenderer` and `BasicMotionPhysics` with all bullets.

<u>`update`</u>: calls the `update` of all bullets, and if a bullet exits the borders of the window it is removed from the list of bullets (do not forget to free the corresponding memory).

<u>`render`</u>: calls the `render` of all bullets.

Testing: create a `StarWarBulletsManager` and test it by calling `shoot` several times in `initGame()`. Later you should remove these calls.

Step 7: `GunInputComponent`

Create an `InputComponent` that represents a gun. The component's constructor receives a `BulletsManger*` and a key to be used for shooting. When the key is pressed it should call method `shoot` of the bullets manager with appropriate values for the initial position and velocity of the bullet as described below.

We assume that `(x,y)` is the top-left corner of the rectangle bounding the fighter's `GamObject`, `w` and `h` are the width and height of that rectangle, `d` is its direction, and `v` is the velocity.

The bullet's initial position should be the middle point in the front of the fighter, this can be calculated as `(x,y)+(w/2,h/2)+d*(h/2)`. The velocity of the bullet should be `N` times the velocity of the fighter, and at least 2 -- try different values for `N`. Recall that the bullet should move in the direction that the fighter is facing, i.e., the bullet's velocity is computed as `d*max(v.magnitute()*N,2.0)`.Note that the magnitude of the direction vector `d` is always 1 -- see `GameObject.cpp`.

Testing: create a `GunInputComponent` controlled by the Space Bar key, and add it to fighter's component.

Step 8: Shots per interval limit

Modify the constructor of `GunInputComponent` to receive two extra parameters: `shotsPerInterval` and `timeInterval` (of type `Uint8` and `Uint32`). The component should allow the player to make at most `shotsPerInterval` shots each `timeInterval` milliseconds.

Testing: modify the `GunInputComponent` that you created in the previous step to make use if these parameters.

Comment: the provided code include a `RendeComponent` called `SkeletonRenderer`, when added to `GameComponent` it draws the bounding rectangle and the velocity and direction vectors. It can help you to debug your program.