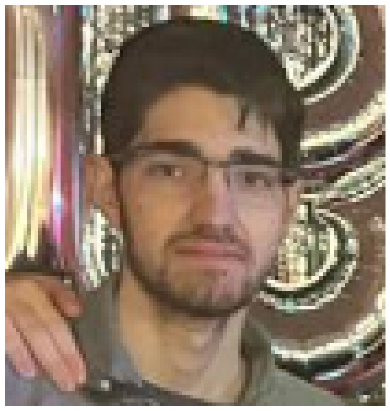
ENTI 3r CDI - Fonaments de modelatge i animació de personatges

Team Description

Members:

Adrià Campos Blanco
Emails: adria.campos@enti.cat / mangaadri@gmail.com

Tomeu Garcia Pintó
Emails: tomeu.garcia@enti.cat / tomeu.garcia.p@gmail.com



NOTICE

dll code found in the directory: /OctopusController

Exercise 1

Ex 1.1

We created a MovingBallTarget.cs script with anOnCollisionEnter(), which notifies MovingBall.cs calling its OnCollisionWithScorpionTail() function.

Inside MovingBall.cs in the OnCollisionWithScorpionTail() is where we tell the MyOctopusController.cs (dll) if it either has to intercept or not the ball with the NotifyShoot() function.

Ex 1.2

Ball no longer moves with WASD, only the goal's blue target does.

Ex 1.3

We added a UI_Controller.sc script (attached at the Camera Canvas), which controls the strength slider. In IK_Scorpion.cs, in UpdateInputs() we call the UI_Controller functions.

IK_Scorpion.cs, in StartShootBall() sets the ball's shootStrengthPer1 based on the strength slider value.

Ex 1.4

In the IKScorpion.cs when we press the Space Key, the ResetLegs() from MyScorpionController.cs (dll) is called, also ResetPosition() from MovingBall.cs.

Ex 1.5

In MovingBall.cs:

- First we compute the shootTimeDuration by lerping 2 arbitrary floats using the shootStrength previously set.
- Then we compute the startVelocity given the startShootPosition, the endPosition (blue target), shootTimeDuration and the acceleration (gravity).

UARM Formula: startVelocity Formula:

Xf = Xo + (Vo*t) + (1/2*a*t^2) -----> Vo = (Xf - Xo - (1/2*a*t^2)) / t

- To compute the instantaneous position and velocity we use the Euler solver formula since in exercise 2 we'll need to apply a Magnus Force (changing acceleration).

Xn+1 = Xn + Vn * dt Vn+1 = Vn + An * dt

Exercise 2

Ex 2.1
IK_Scorpion.cs checks the Z and X input keys and calls UI_Controller.cs UpdateEffectStrengthSlider() method.
IK_Scorpion.cs calls its method ComputeTailTargetPosition() to change ow the tail hits the ball depending on the effect value.

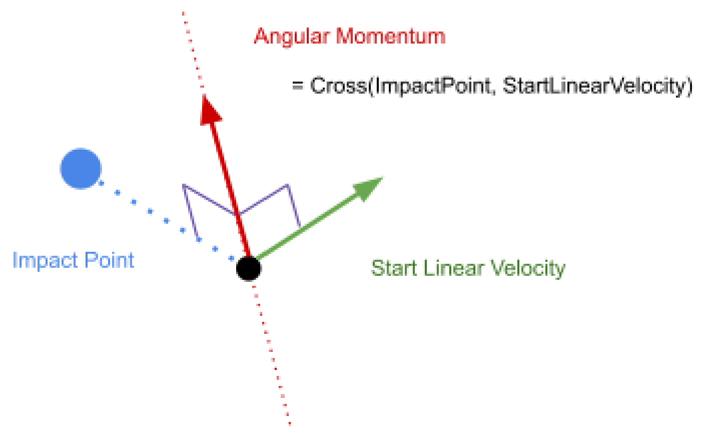
Ex 2.2
MovingBall.cs computes the rotation velocity in ComputeAngularVelocity() and ComputeRotationAxis() to find if the rotation is clockwise (called at Update() if the ball wasn't shot yet).
Then, in the RotateBall() function (called in Update() if the ball was shot) we display the angular velocity in the canvas text.

Ex 2.3
In MovingBall.cs, the method UpdateTrajectoryHints() updates the arrows' transform, and UpdateInputs() checks the I input key to toggle the arrows' visibility.

Ex 2.4
UpdateTrajectoryHints() also calls UpdateBallArrows(), which updates the velocity and forces arrows attached to the ball.

Ex 2.5
Firstly, we compute the impact point on the ball's surface.
Then we compute the angular momentum with the cross product of the impact point with the start linear velocity.
Following, in order to keep the computations cheap and simple, we assume that the torque is equal to the angular momentum.
Finally, we consider that the angular velocity is the torque multiplied by a custom constant (which depends on the effect strength slider) to make the result vary more depending on the effect strength.

```
AngularMomentum = Cross(ImpactPoint, StartLinearVelocity)
Torque = AngularMomentum
AngularVelocity = Torque * Lerp(0, maxEffectStrength, sliderValue)
```

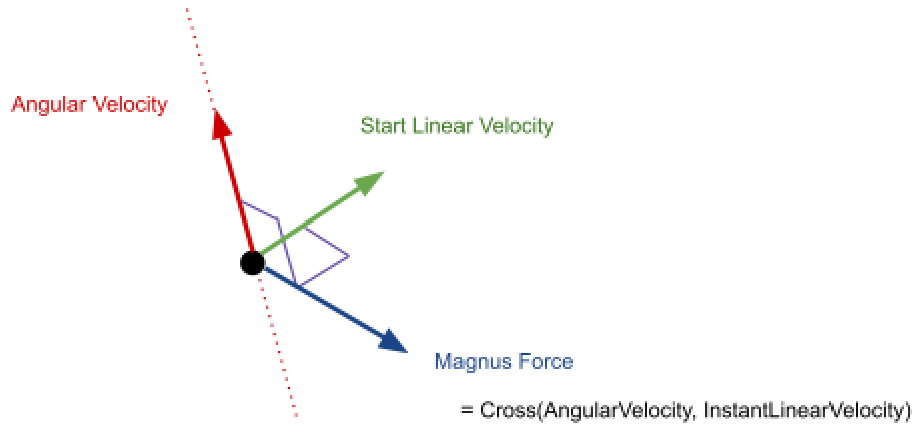


(Computing the Angular Momentum)

Ex 2.6
Each frame after the ball was shot, we need to compute the acceleration, which is compounded by the Gravity and Magnus Effect forces.
The Magnus Force equals the cross product of the angular velocity (constant) and the frame's instantaneous linear velocity.

MagnusForce = Cross(AngularVelocity, InstantLinearVelocity);

We chose this formula because it is the simplest one, which is good enough for our small game. We don't need to take into account the air's density, friction and other physically accurate parameters needed for real physics simulations.



(Computing the Magnus Force)

Exercise 3

Ex 3.1
In the IK_Scorpion.cs script, in the Update(), while the animation is playing, we call the UpdateLegsAndBody() function, which performs the raycasts and updates the future leg bases' positions.
We changed the scorpion prefab so that the Tail and LegTargets game objects are now parented to the Main Body game object.

Ex 3.2
In the scene, we added an "Obstacles" GameObject containing all individual obstacles.

Ex 3.3
In the script MyScorpionController.cs (dll), in the ComputeBaseBonePosition() function, in addition to Lerp() the position we also add a displacement on the Y axis using a sinus (synchronized with the position Lerp()).

Ex 3.4
Back to the UpdateLegsAndBody() function in IK_Scorpion.cs, now we also compute the MainBody's position by adding the average future base leg positions to an initial offset (legs to body height offset).

Ex 3.5
Now in UpdateLegsAndBody() (IK_Scorpion.cs), we also compute separately the average position of the right legs and left leg, with those we find a new RightAxis vector, from which we find the resulting UpAxis used to compute the body's rotation with the function Quaternion.LookRotation().

Ex 3.6
To allow the scorpion to move in different directions facing straight (not walking sideways), we first compute our own forward vector, used to compute the desiredRotation in UpdateLegsAndBody().
Then in the RotateBody() method the desiredRotation is applied progressively with Quaternion.RotateTowards() to make it look smoother.
With this the body faces the move direction but we still have a problem: the future leg bases don't rotate with the body. To fix this we set the Y axis rotation of the base legs holder transform equal to the amount of Euler angles described by the main body's Y axis rotation.

Exercise 4

Ex 4.1

The actual tail target moves around the ball given the effect strength (Ex 2.1).
In addition, we also change the gradient descent learning rate (SetTailLearningRate() function in IK_Scorpion.cs) by lerping between 2 values given the shoot strength. This way the more strength teh faster the tail approaches the target.

Ex 4.2

In MyScorpionController.cs (dll) we added DistanceFromTargetAndOrientation(), a new error function for our gradient descent. This function in addition to returning the end effector distance towards the target also returns an "orientation value" computed from the direction we want the scorpion to hit the ball.

Ex 4.3

We set weights (for the error function distance and orientation) in order to make them vary so that the tail could face the ball the way we want while it keeps approaching the ball. We created an IEnumerator() that changes the weights values while the scorpion moves the tail to the target. First we only give weight to the distance parameter and after only to the orientation.

We had to change the Tail's first bone rotation axis (from Z to Y) to make the orientation actually approach the ball (otherwise it seemed there wasn't any solution to getting closer to the target).
We draw with blue debug lines the ForwardKinematics of the tail, with which we observe an offset to these lines from the actual tail. This error might cause the tail to behave weirdly, making the ForwardKinematics think it is in the wrong way.
Also, the ball doesn't behave correctly (the tail reaches the target flickering) if the ball is shot alternating from left to right.

Exercise 5

Ex 5.1
In order to clamp the angles of our tentacles, first we remove the twist of the bone’s local rotation, and then we clamp the remaining angles on the X and Z axis.

Implemented in the dll script MyOctopusController.cs (dll), in the ClampBoneRotation() function.

Ex 5.2
To make the movement of our tentacles continuous at the moment of stopping the ball. We do a Lerp() between the current position and the blue target position and we set a time in which this movement has to be done. Then we leave the tentacle stopping the ball for 1.5 seconds and when it’s done, we Lerp() back between the blue target position and the movable target by decrementing the lerp timer.

Lerp() is computed in the update_ccd() function (dll script MyOctopusController.cs).

Ex 5.3
To make the tentacles always face the camera, we clamp each tentacle bone between these Euler values: Vector3(-20, 0, -3) and Vector3(20, 0, 3).