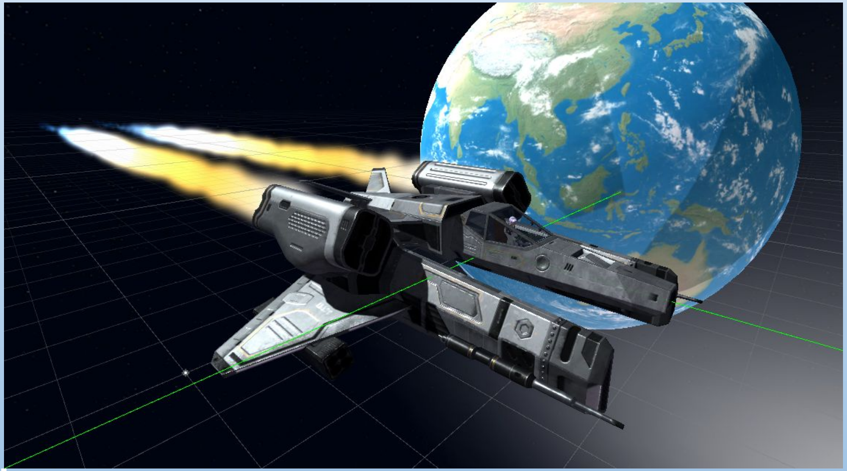
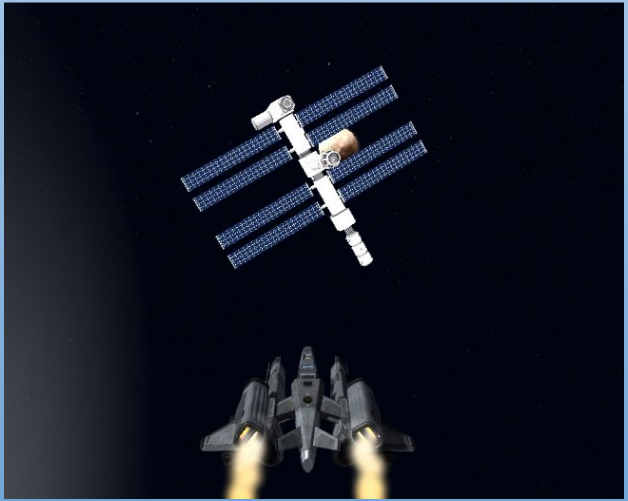
SMART SPACE JOURNEY

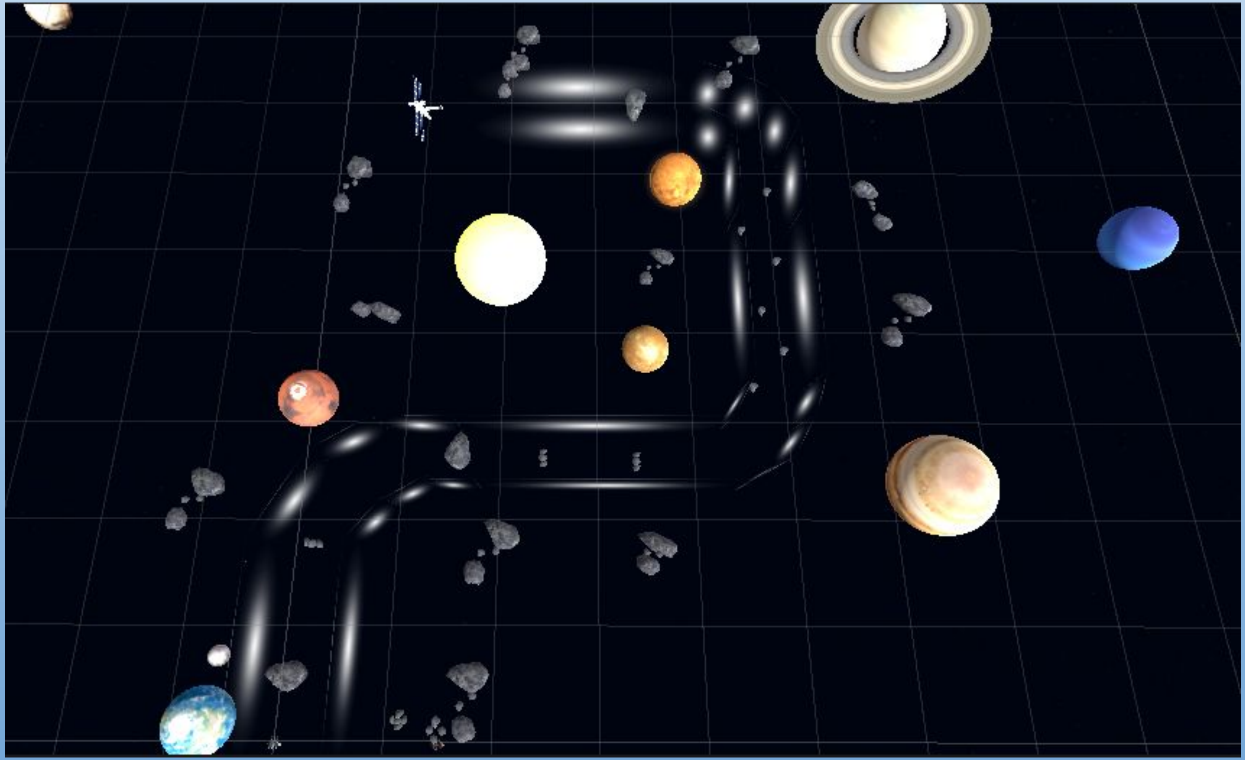
By Luis Miguel García Marín





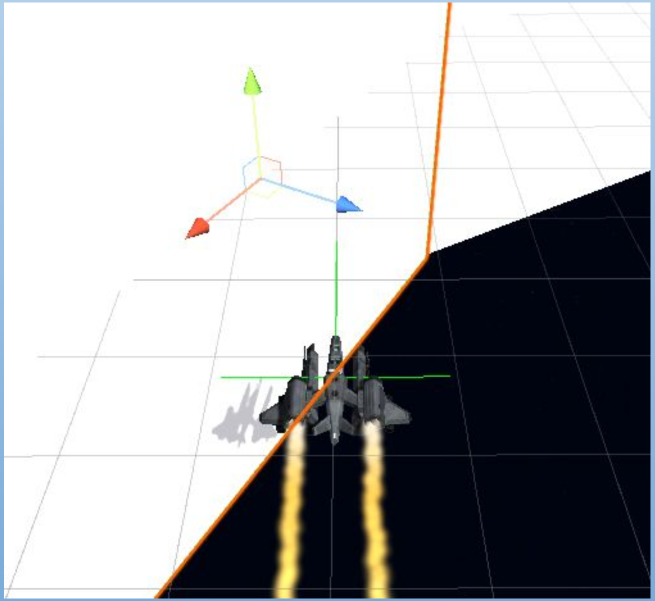
INDEX

JOURNEY



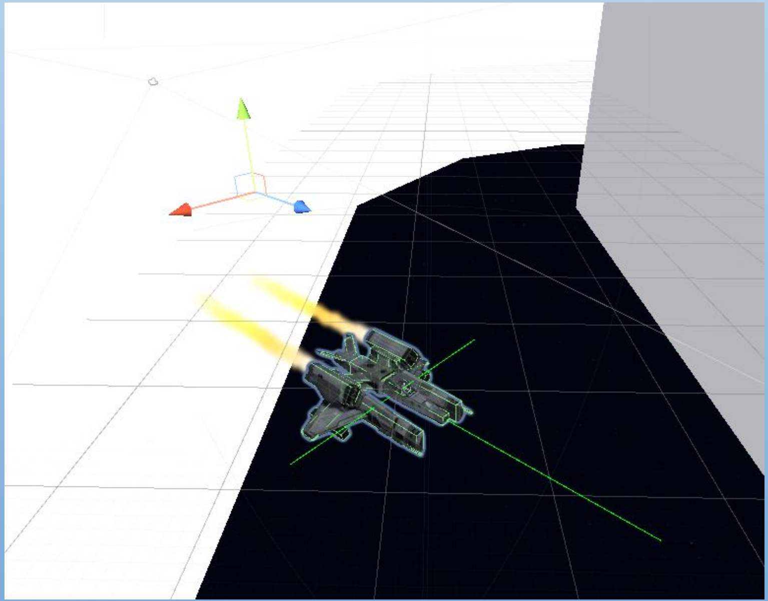
Automatic Navigation AI (Motivation)

- We will consider movement along the X and Z axes for learning purposes. For vertical movement (Y axis), we will use a height-reaching algorithm based on forces.
- <u>We do not use a ground</u> (as it does not exists in space).
- In this way, we find 2 control variables (unknowns):
  - Acceleration
  - Rotation with respect to Y
- Additionally, we have several environmental variables: Initial speed and rotation (as starting points), and the initial and final dot product (with respect to a wall).



Navigation AI (Training)

- This has been done by testing different situations of initial speed and rotation with respect to Y, storing the initial dot product between our forward direction and the normal (forward) of the wall.
- After waiting for a period in which we applied acceleration and rotation, we also stored the resulting dot product after applying the forces.
- The goal will be to <u>minimize</u> the absolute value of the final dot product (to make the forward direction and the normal perpendicular).



Navigation AI (Training)

```
for (float speed = 0; speed <= maximumSpeedValue; speed++)     //Loop for the initial speed
{
   for (float rotation = 0; rotation <= 181; rotation += 5)                    //Loop for the initial rotation
   {
      for (float acceleration = -1; acceleration <= 1.05; acceleration += 0.2f)      //Loop for the acceleration
      {
         for (float turn = -1; turn <= 1.05; turn += 0.2f)                //Loop for the turn
          {
             PlaceShipIdle(rotation);

             rb.velocity = transform.forward * speed; InitialDotProduct = script.dot;
             shipController.speed = acceleration; shipController.turn = turn;
             time = Time.time;

             yield return new WaitUntil(() => Time.time - time >= 0.25);
             FinalDotProduct = script.dot;
             LogCase(speed, InitialDotProduct, acceleration, turn, FinalDotProduct);
}}}}
```

Navigation AI (Experience Table)

Speed	Initial Dot Product	Acceleration	Turn	Final Dot Product
0	-0.998478	-1	-1	-0.997643
0	-0.991598	-1	-0.8	-1.065866
0	-0.760561	-1	-0.6	0
0	-0.996935	-1	-0.4	0.13432
0	-0.990725	-1	-0.2	-0.761051

Navigation AI (With Knowledge)

```
for (float acceleration = -1; acceleration <= 1.05; acceleration += 0.2f)     //Loop for the Fy
{
   testCase = LogVariables(rb.velocity.magnitude, script.dot, acceleration, objectiveDotProduct);
   float turn = (float)predictTurnKnowledge.classifyInstance(testCase);
   if (turn > -0.05 && turn < 0.05) turn = 0; if (turn < -1) turn = -1; if (turn > 1) turn = 1;

        testCase2 = LogVariables2(rb.velocity.magnitude, script.dot, acceleration, turn);
        float finalDotProduct = (float)predictFinalDotProductKnowledge.classifyInstance(testCase2);

        if (Abs(finalDotProduct) < lowestDotProduct || Abs(finalDotProduct) - lowestDotProduct < 0.02)
        {
                SaveSmallestError(Abs(finalDotProduct), acceleration, turn);
        }
}
```

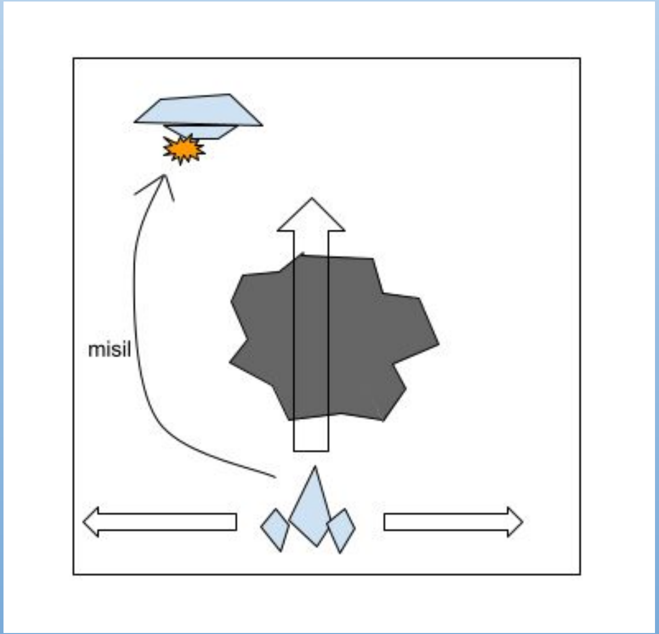
- In each FixedUpdate, we apply the best acceleration and best turn obtained through force control. We do not terminate the state machine so that it continues predicting.

Navigation AI (Apply Values with Physical Control)

```
private void FixedUpdate()
{
        currentAcceleration = acceleration * speed;   // Apply Acceleration
        if (rb.velocity.magnitude < maxSpeed) {
                rb.AddForce(transform.forward * currentAcceleration);
        } else {
                rb.AddForce(-transform.forward * currentAcceleration);
        }
        currentTurnAngle = maxTurnAngle * turn;      // Apply Turn
        if (currentTurnAngle == 0) {
                rb.AddTorque(transform.up * -rb.angularVelocity.y);
        } else {
                rb.AddTorque(transform.up * currentTurnAngle);
}}
```

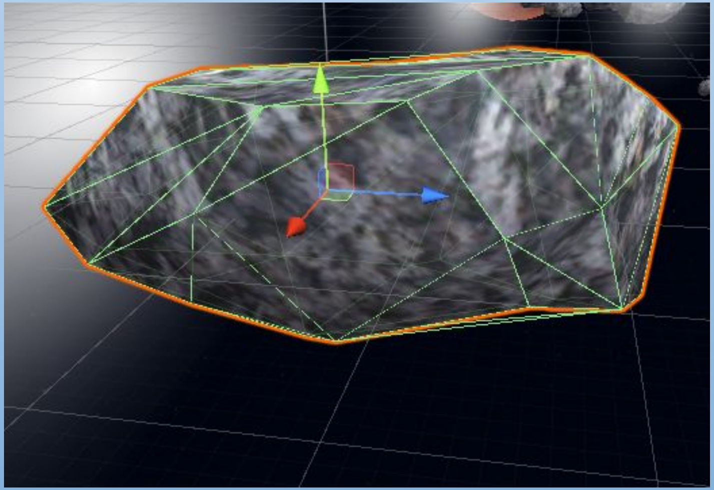
States against Asteroids

- By default, the ship will be in the state of height 0.
- If a large asteroid (with a scale >= 28) is detected via Raycast, it will switch to the ascending state, using the height-reaching algorithm.
- If a group of asteroids connected by FixedJoints is detected via Raycast, it will switch to the shooting state.
- If a small asteroid is detected via Raycast, the ship will attempt to dodge it laterally in the direction where the asteroid is farthest from what is on its left or right.



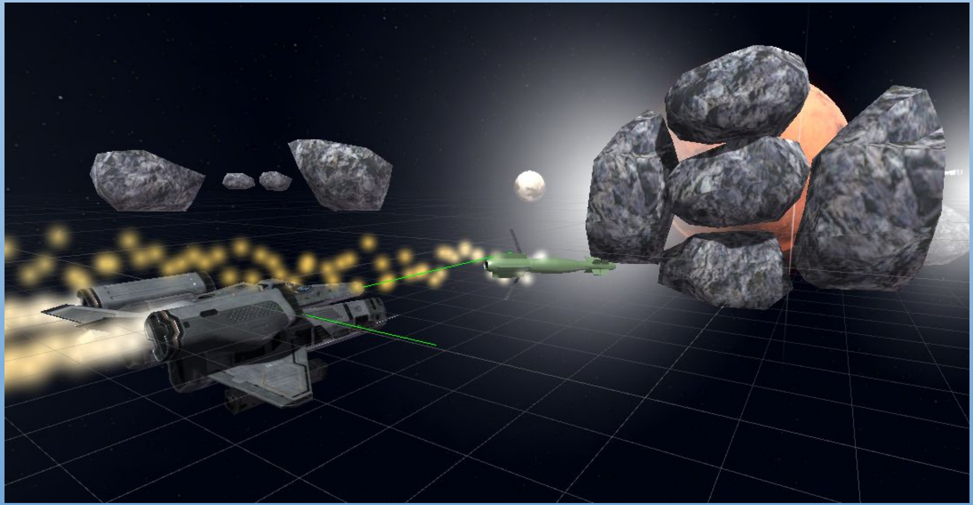
Physical Rotation at Constant Speed of Asteroids (Large and Small)

```
float difference = velocity - rb.angularVelocity.z;
if (rb.angularVelocity.z <= velocity)
{
    float factor = difference * angularSpeed;
    rb.AddTorque(transform.forward * factor);
}
else
{
    float factor = difference * angularSpeed;
    rb.AddTorque(-transform.forward * factor);
}
```



Missile Launch to Break Asteroid Joints

- Upon detecting them via Raycast, the ship will launch a missile propelled by forces towards the group of asteroids. Upon impact, it will apply an ExplosionForce to the asteroids, breaking the FixedJoints. Additionally, fire and sound effects will be produced.



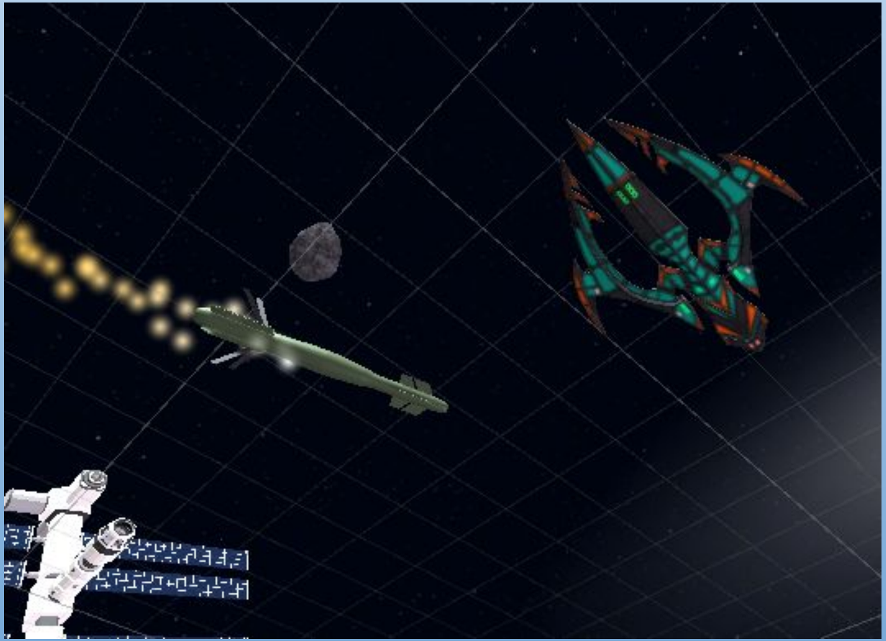
UFO Generation

- At regular intervals, there is a chance (Random.Range(0,2) == 1) that a UFO will be generated, and it will propel itself near us using force control, with a speed limit:

```
if (rb.velocity.magnitude < maxSpeed)
{
    rb.AddForce(transform.forward * module);
}
else
{
    rb.AddForce(-transform.forward * module);
}
```

Guided missile using position and altitude tracking.

- When the ship detects the UFO with a trigger (which can be interpreted as its radar), it fires a guided missile that uses the position tracking and altitude tracking algorithms to pursue the UFO.
- When it collides with the UFO, it will explode and destroy it.



Guided Missile Position Tracking Algorithm

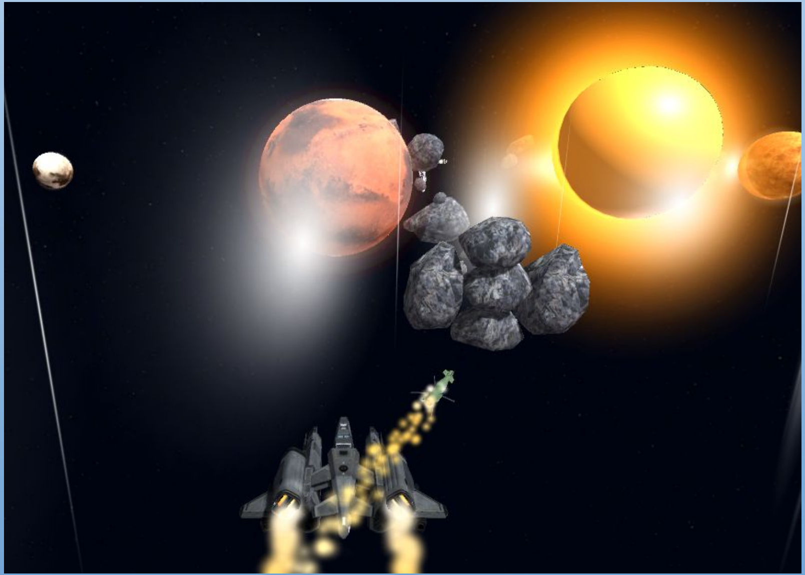
```
private void PositionTracking(Vector3 posObjective, float horizontalSpeed, float frontalPropulsion)
   {
      Vector3 vectorTowardsObjective  = posObjective - transform.position;
      float relativeVelocity = TrackedObject.GetComponent<Rigidbody>().velocity.magnitude - rb.velocity.magnitude;
      float angle = Vector3.Angle(vectorTowardsObjective, GetComponent<Rigidbody>().velocity);
      float factor = vectorTowardsObjective.magnitude * horizontalSpeed;
      if ((relativeVelocity > 0) || (angle < 70))
      {
         rb.AddForce(vectorTowardsObjective * frontalPropulsion * factor);
         rb.transform.LookAt(new Vector3(posObjective.x, rb.transform.position.y, posObjective.z));
      }
      else
      { // Start slowing down...
         // Instead of rb.velocity = rb.velocity * 0.95f; we do it with forces
         rb.AddForce(vectorTowardsObjective * frontalPropulsion * factor); rb.AddForce(-rb.velocity * 2f);
         rb.transform.LookAt(new Vector3(posObjective.x, rb.transform.position.y, posObjective.z));
      }}
```

Animations

- The ship is piloted by an astronaut (a character with a skeleton rig) who has animations for operating the ship's controls.
- She also presses buttons when firing a missile.

Scenery

- In addition to the previously mentioned effects, the scene has been decorated with the planets of the solar system, adding lighting effects, such as in the case of the Sun.



THANK YOU

Luis Miguel García Marín