**Documentation filière recherche :**

**Modélisation 3D du trafic douaisien sous Unity**

Avant de commencer, j’aimerais présenter la méthode actuelle pour lancer le projet. Dans un premier temps, il faut lancer le script python GestionSUMo.py (/Sumo map). Ce script attend qu’une connexion TCP soit établie avant de lancer SUMo et de transmettre les informations de la simulation par la connexion TCP. Il faut ensuite lancer le projet Unity (/Unity/Map\_Douai\_HTC\_Vive). Ce projet va établir la connexion TCP une fois le bouton « Connect » appuyé, suite à quoi il lira les informations envoyées et affichera les véhicules.

**Partie 1 :** Prise en main de Unity

<https://www.raywenderlich.com/147687/introduction-unity-getting-started-part-12>

*Contenu :*

Installation, création d’un premier projet et découverte de l’interface.

<https://www.raywenderlich.com/127672/introduction-unity-scripting>

*Contenu :*

Test et compréhension des Scripts C#.

**Partie 2 :** Prise en main du HTC Vive

<https://www.raywenderlich.com/149239/htc-vive-tutorial-unity>

*Contenu :*

Implémentation du HTC Vive dans Unity et apprentissage des fonctionnalités et du potentiel.

Par la suite, j’ai modifié le fonctionnement du mouvement (Gâchette pour téléportation, grip pour attraper et Touchpad pour se déplacer) et ai étendu la téléportation à toutes les surfaces.

**Partie 3 :** Chargement de la carte avec Mapbox

<https://www.mapbox.com/help/unity-mesh-pt-1>

*Contenu :*

Utiliser Mapbox pour charger une carte à partir des coordonnées.

<https://www.mapbox.com/help/unity-mesh-pt-2/>

*Contenu :*

Extruder les bâtiments en 3D.

Les seuls changements que j’ai effectués sont de mettre les bonnes coordonnées (variable *Latitude Longitude String* de l’objet *Map* fixée à « 50.3667, 3.0667 »), de choisir un zoom de carte permettant de voir suffisamment Douai (variable *Zoom* de l’objet *Map* fixée à « 15 ») et de mettre une taille permettant de se promener sur la carte (variable *Unity Tile Size* de l’objet *Map* fixée à « 550 »). Le FoV limite un peu les options si on prend une vue aérienne.

Un troisième tutoriel est disponible (<https://www.mapbox.com/help/unity-mesh-pt-3/>) qui rentre dans les détails de la mise en forme de style pour les bâtiments. Il est alors possible de personnaliser certains bâtiments important, ou simplement de mettre des textures plus sympathiques que le gris uni disponible par défaut. Les textures peuvent être faites soi-même ou récupérées sur le Unity Asset Store.

**Partie 4 :** TraCI

<http://sumo.dlr.de/wiki/Tutorials/TraCI4Traffic_Lights>

*Contenu :*

Créer une carte simple sous SUMO et contrôler les feux à l’aide de TraCI.

Une fois le fonctionnement global de TraCI compris, j’ai retiré le contrôle des feux, modifié la carte utilisée et ajouté deux fonctions : une qui prépare une liste des véhicules chargés et une qui lit les coordonnées de ces véhicules. Dans la version actuelle, les piétons ne sont pas pris en compte. La partie commentée dans GestionSUMo.py permet de les prendre en compte, mais cette version n’a pas totalement été testée.

**Partie 5 :** Liaison TCP

Rien de particulier à dire : lorsqu’on lance le script python, il attend une connexion sur localhost:5005 avant de lancer SUMO. Tout est paramétrable au début du script (*TCP\_IP* et *TCP\_PORT*). Une fois la connexion établie, un thread se lance qui lit les informations reçues, c’est là qu’on peut récupérer la position de l’utilisateur, mais je n’ai pas trouvé de moyen de créer un objet SUMO absolument libre et de lui forcer des coordonnées. Une solution à envisager est de mettre en place un mode « libre » où il est possible de se promener librement sans être traduit dans SUMo, et pouvoir passer en mode « simulation », où la position serait reportée dans SUMo, mais avec les limites normalement imposées (piéton sur voie piétonne, voiture sur la route), quitte à tricher un peu pour se placer sur la voie la plus proche en cas d’écart.

Côté C#, j’ai récupéré un serveur de base qui permet de lire et afficher les informations reçues et ai juste ajouté une fonction qui découpe le message reçu afin de séparer le nom du véhicule de ses coordonnées. La structure choisie est la suivante : id\_Veh + « : » + coord\_X + « ; » + coord\_Y + « | ». Les coordonnées sont arrondies à deux chiffres après la virgule.

Les informations côté Unity sont ensuite passées dans un script *ObjectCreation* qui cherche un objet ayant pour nom l’id du véhicule, le créé s’il n’en trouve pas et modifie sa position au vu des coordonnées récupérées. Le script pourrait être amélioré pour faire des mouvements plus fluides (cf. annexe 6).

Pour le moment, tous les scripts sont placés dans l’objet *[CameraRig]*. L’objet Socket Script charge automatiquement *TCPConnection*, mais on doit lui fournir *l’ObjectCreation* qui traitera les informations. Le script *ObjectCreation* prend un Objet qui sera l’apparence donnée aux véhicules créés, il serait donc possible de séparer les véhicules par type grâce à leur id et de les afficher différemment (cf annexe 6).

**Partie 6 :** Explication de code (GestionSUMo.py)

Je vais prendre un instant pour découper et expliciter les différentes parties de mon code explicité en annexe. GestionSUMo.py établi la connexion TCP, lance SUMo à l’aide de TraCI et transmet les informations de TraCI à Unity à travers la connexion TCP.

|  |
| --- |
| TCP\_IP = '127.0.0.1'  TCP\_PORT = 5005  BUFFER\_SIZE = 1024  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  s.bind((TCP\_IP, TCP\_PORT)) |

*Informations TCP (GestionSUMo.py, lignes 13 à 18)*

Après les imports, on peut modifier les informations importantes de la liaison TCP : l’adresse (*TCP\_IP*), le port (*TCP\_PORT*) ou la taille du buffer (*BUFFER\_SIZE*).

|  |
| --- |
| class SocketThread(Thread):  def \_\_init\_\_(self, conn):  Thread.\_\_init\_\_(self)  self.conn = conn  self.\_isRunning = True  def run(self):  while self.\_isRunning:  data = conn.recv(BUFFER\_SIZE).decode()  if data != "":  # This is where the informations about the user are to be used in SUMo  print("received data:", data)  if data == "Fin\r":  conn.close()  self.stop()  def stop(self):  self.\_isRunning = False  def send(self, message):  conn.send(message. encode("utf-8")) |

*SocketThread (GestionSUMo.py, lignes 22 à 42)*

Pour le moment, le thread se contente d’afficher les informations reçues depuis Unity, mais c’est par là qu’il sera possible de connaitre la position de l’utilisateur et mettre à jour la simulation pour le prendre en compte.

Il suffit d’appeler la fonction *send(message)* du *SocketThread* pour transmettre le String *message* à Unity, qui décode le message et le découpe selon des règles expliquées plus loin.

Le paramètre *\_isRunning* permet de mettre fin à la boucle une fois la connexion finie, ce qui est fait à la réception du message « Fin\r » en passant par la fonction *stop(self)*.

|  |
| --- |
| s.listen(1)  conn, addr = s.accept()  print('Connection address:', addr)  threadUnity = SocketThread(conn)  threadUnity.start() |

*Connexion TCP (GestionSUMo.py, lignes 46 à 50)*

Dans un premier temps, on attend qu’une connexion TCP soit établie pour ensuite créer un *SocketThread* contenant les informations de cette connexion. De cette façon, on pourra surveiller tout au long de la simulation ce qu’Unity nous transmet pour mettre à jour l’agent au sein de la simulation.

|  |
| --- |
| step = 0  vehicles = []  while traci.simulation.getMinExpectedNumber() > 0: |

*Début de la boucle TraCI (GestionSUMo.py, lignes 55 à 57)*

La variable *step* compte le nombre de pas de simulation écoulés, elle peut être utilisée si on souhaite mettre fin à la simulation au bout d’un certain temps. Dans notre cas, la condition d’arrêt est « traci.simulation.getMinExpectedNumber() <= 0 », autrement dit, que la somme des véhicule dans la simulation et encore à venir dans la simulation soit nulle.

La liste *vehicles* contiendra l’intégralité des ID des véhicules chargés dans la simulation, comme il sera expliqué sous peu. Je n’ai pas trouvé de manière plus simple permettant de connaître les ID des véhicules, qui sont utilisés plus loin afin de récupérer leurs coordonnées et les afficher correctement dans Unity.

|  |
| --- |
| if len(traci.simulation.getLoadedIDList()) > 0:  vehicles = vehicles + traci.simulation.getLoadedIDList()  if len(traci.simulation.getArrivedIDList()) > 0:  vehicles = [x for x in vehicles if x not in traci.simulation.getArrivedIDList()] |

*Mise à jour de la liste vehicles (GestionSUMo.py, lignes 61 à 66)*

La liste *vehicles* se fait en deux parties. Dans un premier temps, on regarde si des véhicules ont étés chargés durant ce pas de simulation. Si c’est le cas, on les ajoute à la liste. Dans un second temps, on regarde si des véhicules sont arrivés durant ce pas de simulation, et donc retirés. Si c’est le cas, on les retire de la liste *vehicles*. De cette façon, on ne transmet pas les informations des véhicules qui n’ont plus d’importance, sans oublier de charger des véhicules qui pourraient être ajoutés plus tard.

Dans le cas de l’exemple qui m’a été fourni, tous les véhicules étaient chargés dès le début, mais mieux vaut prévenir que guérir. Il est à noter que les piétons n’apparaissent pas comme des véhicules et ne sont donc pas transférés correctement à Unity, mais je n’ai pas trouvé de moyen des récupérer leurs IDs.

|  |
| --- |
| for veh in vehicles:  print(traci.vehicle.getPosition(veh))  # The message is in the format "id:x;y|". The | marks the separation between two vehicles.  message = veh + ":" + str(round(traci.vehicle.getPosition(veh)[0], 2)) + ";" + str(round(traci.vehicle.getPosition(veh)[1], 2)) + "|"  threadUnity.send(message) |

*Transmission des coordonnées (GestionSUMo.py, lignes 70 à 74)*

Pour chaque véhicule dans *vehicles*, on récupère ses coordonnées et on les envoie à travers le *SocketThread* en suivant la codification déjà évoquée (id\_Veh + « : » + coord\_X + « ; » + coord\_Y + « | »).

|  |
| --- |
| def get\_options():  optParser = optparse.OptionParser()  optParser.add\_option("--nogui", action="store\_true",  default=False, help="run the commandline version of sumo")  options, args = optParser.parse\_args()  return options |

*Parser --nogui (GestionSUMo.py, lignes 81 à 86)*

On ajoute l’option de lancer la simulation SUMo sans interface graphique.

|  |
| --- |
| traci.start([sumoBinary, "-c", "Project\_orio.sumocfg"]) |

*Lancement de TraCI (GestionSUMo.py, lignes 102)*

C’est là qu’est défini le projet SUMo lancé dans la simulation. *sumoBinary* est défini en fonction du parser défini plus tôt.

Annexes

**Annexe 1 :** GestionSUMo.py

Ce script a déjà été présenté et expliqué.

|  |
| --- |
| from \_\_future\_\_ import absolute\_import  from \_\_future\_\_ import print\_function  import optparse  import socket  from threading import Thread  from sumolib import checkBinary  import traci  import traci.constants as tc  #TCP informations  TCP\_IP = '127.0.0.1'  TCP\_PORT = 5005  BUFFER\_SIZE = 1024  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  s.bind((TCP\_IP, TCP\_PORT))  #Threads to check for informations from the Unity Simulation  class SocketThread(Thread):  def \_\_init\_\_(self, conn):  Thread.\_\_init\_\_(self)  self.conn = conn  self.\_isRunning = True  def run(self):  while self.\_isRunning:  data = conn.recv(BUFFER\_SIZE).decode()  if data != "":  # This is where the informations about the user are to be used in SUMo  print("received data:", data)  if data == "Fin\r":  conn.close()  self.stop()  def stop(self):  self.\_isRunning = False  def send(self, message):  conn.send(message.encode("utf-8"))  # Listening for connection from the Unity Socket  s.listen(1)  conn, addr = s.accept()  print('Connection address:', addr)  threadUnity = SocketThread(conn)  threadUnity.start()  def run():  """execute the TraCI control loop"""  step = 0  vehicles = []  # persons = []  while traci.simulation.getMinExpectedNumber() > 0:    # Adding the new vehicle to the list of vehicles  if len(traci.simulation.getLoadedIDList()) > 0:  vehicles = vehicles + traci.simulation.getLoadedIDList()  # Removing from the list of vehicles those who were removed  if len(traci.simulation.getArrivedIDList()) > 0:  vehicles = [x for x in vehicles if x not in traci.simulation.getArrivedIDList()]  """  Another solution would be to load the vehicles in a list and the persons in another. That way, the persons are taken in account. This is to work on.  # Adding the vehicles to the list  if len(traci.vehicle.getIDList()) > 0:  vehicles = traci.vehicle.getIDList()  # Adding the persons to the list  if len(traci.person.getIDList()) > 0:  persons = traci.person.getIDList()  """    print("Time:")  print(traci.simulation.getCurrentTime())  for veh in vehicles:  print(traci.vehicle.getPosition(veh))  # The message is in the format "id:x;y|". The | marks the separation between two vehicles.  message = veh + ":" + str(round(traci.vehicle.getPosition(veh)[0], 2)) + ";" + str(round(traci.vehicle.getPosition(veh)[1], 2)) + "|"  threadUnity.send(message)  """  for per in persons:  print(traci.person.getPosition(per))  # The message is in the format "id:x;y|". The | marks the separation between two vehicles.  message = per + ":" + str(round(traci.vehicle.getPosition(veh)[0], 2)) + ";" + str(round(traci.vehicle.getPosition(veh)[1], 2)) + "|"  threadUnity.send(message)  """  traci.simulationStep()  step += 1  traci.close()  sys.stdout.flush()  def get\_options():  optParser = optparse.OptionParser()  optParser.add\_option("--nogui", action="store\_true",  default=False, help="run the commandline version of sumo")  options, args = optParser.parse\_args()  return options  # this is the main entry point of this script  if \_\_name\_\_ == "\_\_main\_\_":  options = get\_options()  # this script has been called from the command line. It will start sumo as a  # server, then connect and run  if options.nogui:  sumoBinary = checkBinary('sumo')  else:  sumoBinary = checkBinary('sumo-gui')  # this is the normal way of using traci. sumo is started as a  # subprocess and then the python script connects and runs  traci.start([sumoBinary, "-c", "Project\_orio.sumocfg"])  run() |

**Annexe 2 :** ViveControllerInputTest.cs

Ce script n’est plus utilisé, il a été mis en place lors du tutoriel de la partie 2 et permet de tester toutes les fonctionnalités des contrôleurs HTC Vive.

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  public class ViveControllerInputTest : MonoBehaviour {    private SteamVR\_TrackedObject trackedObj;  private SteamVR\_Controller.Device Controller  {  get { return SteamVR\_Controller.Input((int)trackedObj.index); }  }  void Awake()  {  trackedObj = GetComponent<SteamVR\_TrackedObject>();  }  // Ce script permet d'afficher dans les logs les actions qui sont réalisées afin de tester les contrôleurs HTC Vive  void Update () {  if (Controller.GetAxis() != Vector2.zero)  {  Debug.Log(gameObject.name + Controller.GetAxis());  }    if (Controller.GetHairTriggerDown())  {  Debug.Log(gameObject.name + " Trigger Press");  }    if (Controller.GetHairTriggerUp())  {  Debug.Log(gameObject.name + " Trigger Release");  }    if (Controller.GetPressDown(SteamVR\_Controller.ButtonMask.Grip))  {  Debug.Log(gameObject.name + " Grip Press");  }    if (Controller.GetPressUp(SteamVR\_Controller.ButtonMask.Grip))  {  Debug.Log(gameObject.name + " Grip Release");  }  }  } |

**Annexe 3 :** ControllerGrabObject.cs

Ce script est tiré du même tutoriel que le précédent et a été légèrement modifié afin d’utiliser les boutons latéraux pour soulever des objets, ce qui me semblait plus logique, ainsi que l’ajout du déplacement par touchpad, qui est pratique une fois maîtrisé. La vitesse du « déplacement touchpad » peut être réglée dans l’Object Unity.

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  public class ControllerGrabObject : MonoBehaviour {  public GameObject cameraRig;  public float acceleration;  private float speed;  private SteamVR\_TrackedObject trackedObj;  // The GameObject that the trigger is currently colliding with  private GameObject collidingObject;  // Reference to the GameObject the player is currently holding  private GameObject objectInHand;  private SteamVR\_Controller.Device Controller  {  get { return SteamVR\_Controller.Input((int)trackedObj.index); }  }  void Awake()  {  trackedObj = GetComponent<SteamVR\_TrackedObject>();  }  // Checks if the user is holding an object, otherwise change the collidingObject  private void SetCollidingObject(Collider col)  {  if (collidingObject || !col.GetComponent<Rigidbody>())  {  return;  }  collidingObject = col.gameObject;  }  // When the trigger collides with an object, calls the method SetCollidingObject(Collider col)  public void OnTriggerEnter(Collider other)  {  SetCollidingObject(other);  }    // Keeps the good object when it is held  public void OnTriggerStay(Collider other)  {  SetCollidingObject(other);  }  // Resets the collidingObject when it is not in contact anymore  public void OnTriggerExit(Collider other)  {  if (!collidingObject)  {  return;  }  collidingObject = null;  }  // Sets the objectInHand, resets the collidingObject and fixes the object to the controller  private void GrabObject()  {  objectInHand = collidingObject;  collidingObject = null;  var joint = AddFixedJoint();  joint.connectedBody = objectInHand.GetComponent<Rigidbody>();  }    // Fixes the object with enough force so it doesn't break  private FixedJoint AddFixedJoint()  {  FixedJoint fx = gameObject.AddComponent<FixedJoint>();  fx.breakForce = 20000;  fx.breakTorque = 20000;  return fx;  }  // Deletes the joint and resets the objectInHand  private void ReleaseObject()  {  if (GetComponent<FixedJoint>())  {  GetComponent<FixedJoint>().connectedBody = null;  Destroy(GetComponent<FixedJoint>());  objectInHand.GetComponent<Rigidbody>().velocity = Controller.velocity;  objectInHand.GetComponent<Rigidbody>().angularVelocity = Controller.angularVelocity;  }  objectInHand = null;  }    void Update () {  // Calls the grab function when the grip buttons are pressed  if (Controller.GetPressDown(SteamVR\_Controller.ButtonMask.Grip))  {  if (collidingObject)  {  GrabObject();  }  }  // Calls the release function when the grip buttons are released  if (Controller.GetPressUp(SteamVR\_Controller.ButtonMask.Grip))  {  if (objectInHand)  {  ReleaseObject();  }  }  // Pushes the cameraRig according to the position of the finger on the touchpad. The speed can be changed.  if (Controller.GetAxis() != Vector2.zero)  {  Debug.Log(gameObject.name + Controller.GetAxis());  if (Controller.GetPress(SteamVR\_Controller.ButtonMask.Touchpad))  {  speed = acceleration \* 5;  } else  {  speed = acceleration;  }  Vector3 newPosition = cameraRig.transform.position;  newPosition += Controller.GetAxis().y \* transform.forward \* speed \* Time.deltaTime;  newPosition += Controller.GetAxis().x \* transform.right \* speed \* Time.deltaTime;  cameraRig.transform.position = newPosition;  }  }  } |

**Annexe 4 :** SocketScript.cs

Ce script appelle les scripts TCPConnection.cs et ObjectCreation.cs, faisant le lien entre les deux.

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System.Collections.Generic;  using System.Text;  using System;  using System.Linq;  public class SocketScript : MonoBehaviour  {  // Variables  private TCPConnection myTCP;  public ObjectCreation Creation;  private string serverMsg;  public string msgToServer;  void Awake()  {  // Adds a copy of TCPConnection to this game object  myTCP = gameObject.AddComponent<TCPConnection>();  }    void Start()  {    }    void Update()  {  // Keeps checking the server for messages, if a message is received from server, it gets logged in the Debug console (see function below)  SocketResponse();  }    void OnGUI()  {  // If no connection has been made, displays button to connect  if (myTCP.socketReady == false)  {    if (GUILayout.Button("Connect"))  {  // Tries to connect  Debug.Log("Attempting to connect..");  myTCP.setupSocket();  }  }    // Once a connection has been made, displays editable text field with a button to send that string to the server (see function below)  if (myTCP.socketReady == true)  {  msgToServer = GUILayout.TextField(msgToServer);  if (GUILayout.Button("Write to server", GUILayout.Height(30)))  {  SendToServer(msgToServer);  }  }  }  // Socket reading script  void SocketResponse()  {  string serverSays = myTCP.readSocket();  if (serverSays != "")  {  // Spliting the String to seperate the vehicles. The last element from temp is removed as it is always EOF.  String[] temp = serverSays.Split('|');  String[] elements = new String[temp.Length - 1];  Array.Copy(temp, elements, temp.Length - 1);  foreach (String element in elements)  {  if (!element.Equals(""))  Creation.UpdateVehicule(element);  }  }  }    // Sends a message to the server  public void SendToServer(string str)  {  myTCP.writeSocket(str);  Debug.Log("[CLIENT] -> " + str);  }  } |

**Annexe 5 :** TCPConnection.cs

Ce script n’est directement ajouté dans aucun objet Unity, il est appelé par SocketScript et contient juste la création de la connexion TCP, la lecture et l’écriture de messages. C’est là qu’il faut paramétrer la connexion.

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  using System.IO;  using System.Net.Sockets;  public class TCPConnection : MonoBehaviour  {  // The name of the connection, not required but better for overview if you have more than 1 connections running  public string conName = "Localhost";  // ip address of the server, 127.0.0.1 is for your own computer  public string conHost = "127.0.0.1";  // Port for the server, make sure to unblock this in your router firewall if you want to allow external connections  public int conPort = 5005;  // a true/false variable for connection status  public bool socketReady = false;  TcpClient mySocket;  NetworkStream theStream;  StreamWriter theWriter;  StreamReader theReader;  // Tries to initiate connection  public void setupSocket()  {  try  {  mySocket = new TcpClient(conHost, conPort);  theStream = mySocket.GetStream();  theWriter = new StreamWriter(theStream);  theReader = new StreamReader(theStream);  socketReady = true;  }  catch (Exception e)  {  Debug.Log("Socket error:" + e);  }  }  // Sends a message to server  public void writeSocket(string theLine)  {  if (!socketReady)  return;  String tmpString = theLine + "\r";  theWriter.Write(tmpString);  theWriter.Flush();  }  // Reads the message from server  public string readSocket()  {  String result = "";  if (theStream.DataAvailable)  {  Byte[] inStream = new Byte[mySocket.SendBufferSize];  theStream.Read(inStream, 0, inStream.Length);  result += System.Text.Encoding.UTF8.GetString(inStream);  }  return result;  }  // Disconnects from the socket (Not used)  public void closeSocket()  {  if (!socketReady)  return;  theWriter.Close();  theReader.Close();  mySocket.Close();  socketReady = false;  }  // Keeps connection alive, reconnect if connection lost  public void maintainConnection()  {  if (!theStream.CanRead)  {  setupSocket();  }  }  } |

**Annexe 6 :** ObjectCreation.cs

Afin de créer les véhicules, ce script est appelé par SocketScript.cs. J’ai déjà évoqué une amélioration possible, à savoir identifier les différents types de véhicules grâce à leur IDs et les afficher différemment. Pour cela, il faut créer différentes variables GameObject comme vehicule dans le script, créer des objets Unity adaptés et les ajouter depuis l’inspecteur de l’élément [CameraRig]. Idéalement, ces objets seront stockés dans le dossier Prefabs qui existe à cet effet.

Ce script impose aussi aux objets les coordonnées pour imiter le déplacement prévu par SUMo. Il serait possible de remplacer ce système par un déplacement linéaire avec une fonction Transform.translate (<https://docs.unity3d.com/ScriptReference/Transform.Translate.html>) ou en utilisant un Vector3.Lerp (<https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html>) qui rendrait le mouvement un peu plus fluide.

|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using System.Globalization;  using UnityEngine;  public class ObjectCreation : MonoBehaviour {  public GameObject vehicule;  public float zoom;  public int x\_shift;  public int y\_shift;  // Use this for initialization  void Start () {  }    // Update is called once per frame  void Update () {  }  public void UpdateVehicule (string info)  {  // First, we split the String to seperate the informations.  string name = info.Split(':')[0];  Debug.Log(name);  string coordinates = info.Split(':')[1];  Debug.Log(coordinates);  // Then, the position is calculated, accounting for the size of the map.  Vector3 position = new Vector3((float) (float.Parse(coordinates.Split(';')[0], CultureInfo.InvariantCulture.NumberFormat)\*zoom+x\_shift), 0, (float) (float.Parse(coordinates.Split(';')[1], CultureInfo.InvariantCulture.NumberFormat)\*zoom+ y\_shift));  // We look if the object already exists, creating it if not.  GameObject Temp = GameObject.Find(name);  if (Temp == null)  {  Temp = Instantiate(vehicule, position, new Quaternion());  Temp.name = name;  }  // Finally, the new coordinates are applied.  Temp.transform.position = position;  }  } |