Annexe : Ensemble des scripts codés « à la main »

Des fonctions des bibliothèques publiques suivantes ont été utilisées :

- *System*
- *System.Collection*
- *System.Collection.Generic*
- *Mathf*
- *Application*
- *UnityEngine*
- *UnityEngine.UI*
- *Physics*
- *Input*
- *Debug*

Liste des scipts

Bipoint 2

Caméra Manager 4

CaterpillarMover 5

GraphDisplay 6

Lidar 9

ObjectDetector 12

Obstacle 12

OccupancyMap 16

PingManager 18

RayInfo 18

Quadrillage 20

RobotController 23

TitleScreen 24

Bipoint

(structure correspondant à deux vecteurs en dimension 3)

```
using UnityEngine;

public struct Bipoint
{
    public Vector3 origine;
    public Vector3 flèche;

    //On définit le Bipoint, composé d'un Vector3 de départ et un Vector3 d'arrivée
    public Bipoint(Vector3 origine, Vector3 flèche, bool usingflèche = true)
    {
        this.origine = origine;
        this.flèche = usingflèche ? flèche : origine + flèche;
    }

#pragma warning disable IDE1006
    public static Bipoint zero
    {
        get => new Bipoint(Vector3.zero, Vector3.zero);
    }

    //Renvoit le Vector3 direction de ce Bipoint
    public Vector3 direction
    {
        get => this.flèche - this.origine;
        set => this.flèche = this.origine + this.direction;
    }

    //Renvoit le flottant de la norme de la direction de ce Bipoint
    public float magnitude
    {
        get => this.direction.magnitude;
    }
#pragma warning restore IDE1006

    //Renvoit un bipoint de même origine dont la direction a pour norme 1
    public Bipoint Normalize()
    {
        Bipoint normalizedBipoint = new Bipoint
        {
            origine = this.origine,
            direction = this.direction.normalized
        };
        return normalizedBipoint;
    }

    //Copie le Bipoint
    public Bipoint Copy()
    {
        return new Bipoint(origine, flèche);
    }

    //Sert à renvoyer un Bipoint en String pour le deboggage
    public override string ToString()
    {
        return "Bipoint : (" + origine.ToString() + "), (" + flèche.ToString() + "))";
    }

    //Transforme un Bipoint en Ray
```

```
    public Ray ToRay()
    {
        return new Ray(this.origine, this.direction);
    }

    //On définit les relations de comparaisons entre Bipoints
    public static bool operator ==(Bipoint left, Bipoint right)
    {
        return (right.origine == left.origine && right.flèche == left.flèche);
    }
    public static bool operator !=(Bipoint left, Bipoint right)
    {
        return (right.origine != left.origine || right.flèche != left.flèche);
    }

    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}
```

Caméra Manager

(gestionnaire des caméras)

```
using UnityEngine;

public class CameraManager : MonoBehaviour
{
    public Camera[] cameras;
    public int currentCamera;
    public KeyCode nextKey;

    void Start()
    {
        UpdateEnabled();
    }

    void Update()
    {
        if (Input.GetKeyDown(nextKey))
        {
            currentCamera = (currentCamera + 1) % cameras.Length;
            UpdateEnabled();
        }
    }

    void UpdateEnabled()
    {
        for (int i = 0; i < cameras.Length; ++i)
        {
            cameras[i].enabled = (i == currentCamera);
        }
    }
}
```

CaterpillarMover

(gère la physique appliquée à une chaîne du robot)

```
using UnityEngine;

public class CaterpillarMover : MonoBehaviour
{
    private Rigidbody robotRigidbody;
    private Collider whCollider;
    private RobotController robotController;

    public Vector3 normalVector;
    private Vector3 wheelsRotationVector;
    private float powerMax;
    private float turnRate;
    private float rearRate;

#pragma warning disable IDE0051
    // Start is called before the first frame update
    void Start()
    {
        robotRigidbody = gameObject.GetComponentInParent<Rigidbody>();
        whCollider = gameObject.GetComponent<Collider>();
        robotController = gameObject.GetComponentInParent<RobotController>();

        powerMax = robotController.powerMax;
        turnRate = robotController.turnRate;
        rearRate = robotController.rearRate;
    }

    // Update is called once per frame
    void Update()
    {
        wheelsRotationVector = Vector3.Cross(normalVector,
robotRigidbody.transform.forward);
    }
#pragma warning restore IDE0051

    // Applique au point de contact voulu une force power
    public void Move(ContactPoint contact, Side side, Color color)
    {
        float power = powerMax * (robotController.inputY + turnRate * (int)side *
robotController.inputX);

        if (power < 0) { power *= rearRate; }

        if (whCollider == contact.thisCollider)
        {
            Vector3 tractionForceAtContact = Vector3.Cross(wheelsRotationVector,
contact.normal) * power;
            robotRigidbody.AddForceAtPosition(tractionForceAtContact, contact.point,
ForceMode.Force);

            Debug.DrawRay(contact.point, tractionForceAtContact, color);
        }
    }
}
```

GraphDisplay

(lit les données de occupancyMap pour les afficher sur une texture)

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

public class GraphDisplay : MonoBehaviour
{
    private OccupancyMap occupancyMap;
    public PingManager pingManager;

    public RawImage oMapDisp;
    public RawImage dMapDisp;
    public RawImage cMapDisp;

    public Color32[] colors;

    public KeyCode update;
    public KeyCode coucheSup;
    public KeyCode coucheInf;
    public KeyCode agrOMap;
    public KeyCode agrDMap;

    public Vector2 oMapOffsetMax;
    public Vector2 oMapOffsetMin;
    public Vector3 oMapSideScale;

    public Vector3 oMapCenterPos;
    public Vector3 oMapCenterScale;

    public Vector2 dMapOffsetMax;
    public Vector2 dMapOffsetMin;
    public Vector3 dMapSideScale;

    public Vector3 dMapCenterPos;
    public Vector3 dMapCenterScale;

    private Texture2D[] graph;
    private int nbCouches;
    private int layer = -1;

    public int nbPerFrame;

    private Texture2D depthMap;

    public float colorDispTime;

    void Start()
    {
        occupancyMap = gameObject.GetComponent<OccupancyMap>();

        nbCouches = occupancyMap.size.y;

        graph = new Texture2D[nbCouches];

        for (int i = 0; i < nbCouches; i++)
        {
            graph[i] = new Texture2D(occupancyMap.size.z + 1, occupancyMap.size.x + 1,
TextureFormat.RGB24, false);
        }
```

```
        ChangeLayer(true);
    }

    private void Update()
    {

        if (Input.GetKeyDown(update))
        {
            StartCoroutine(UpdateOccupationMap());
        }

        if (Input.GetKeyDown(coucheSup))
        {
            ChangeLayer(true);
        }
        else if (Input.GetKeyDown(coucheInf))
        {
            ChangeLayer(false);
        }

        if (Input.GetKeyDown(agrOMap))
        {
            EnlargeImage(oMapDisp, oMapCenterPos, oMapCenterScale);
        }
        else if (Input.GetKeyUp(agrOMap))
        {
            ReduceImage(oMapDisp, oMapOffsetMax, oMapOffsetMin, oMapSideScale);
        }

        if (Input.GetKeyDown(agrDMap))
        {
            EnlargeImage(dMapDisp, dMapCenterPos, dMapCenterScale);
        }
        else if (Input.GetKeyUp(agrDMap))
        {
            ReduceImage(dMapDisp, dMapOffsetMax, dMapOffsetMin, dMapSideScale);
        }
    }

    //Lit la carte de OccupancyMap pour la transposer dans la texture
    //On utilise une Coroutine pour éviter de faire lagger la simulation
    public IEnumerator UpdateOccupationMap()
    {
        Debug.Log("called");
        for (int couche = 0; couche < nbCouches; couche++)
        {
            Texture2D texture = graph[couche];
            texture.filterMode = FilterMode.Point;

            for (int z = 0; z < texture.height; z++)
            {
                for (int x = 0; x < texture.width; x++)
                {
                    texture.SetPixel(x, texture.height - z,
colors[(int)occupancyMap.carte[z, x, couche]]);

                    //Tous les nbPerFrame points calculés, on change de frame pour
éviter que la simulation ne ralentisse
                    if (((z + 1) * (x + 1)) % nbPerFrame == 0)
                    {
                        yield return null;
                        Debug.Log($"{x},{z}");
```

```
                }
            }
        }

        texture.Apply(false);

        byte[] image = texture.EncodeToPNG();
        //File.WriteAllBytes($"couche {couche}.png", image);

        Debug.Log($"layer {couche} has been updated !");
    }
    Debug.Log("The map has been updated !");
    yield return null;
}

private void ChangeLayer(bool monter)
{
    layer = monter ? layer + 1 : layer + nbCouches - 1;
    layer %= nbCouches;

    graph[layer].Apply();
    oMapDisp.texture = graph[layer];

    Debug.Log($"displayed layer : {layer}");
}

private void EnlargeImage(RawImage image, Vector3 pos, Vector3 scale)
{
    image.transform.localPosition = pos;
    image.transform.localScale = scale;
}

private void ReduceImage(RawImage image, Vector2 offsetMax, Vector2 offsetMin,
Vector3 scale)
{
    image.transform.localScale = scale;
    image.rectTransform.offsetMax = offsetMax;
    image.rectTransform.offsetMin = offsetMin;
}


public void UpdateDepthMap(float[,] depthTable, float viewDistance)
{
    depthMap = Lidar.EncodeDepthMap(depthTable, viewDistance);
    DisplayDepthMap(depthMap);
}

private void DisplayDepthMap(Texture2D heightMap)
{
    dMapDisp.texture = heightMap;
}

public IEnumerator DispColorMap(Texture2D colorMap)
{
    cMapDisp.texture = colorMap;
    cMapDisp.enabled = true;

    yield return new WaitForSeconds(colorDispTime);

    cMapDisp.enabled = false;
}
}
```

Lidar

(classe générant un système ressemblant au lidar, notemmant en envoyant des rayons et en lisant les données)

```
using UnityEngine;

public class Lidar
{
    //On envoie des rayons parallèles au sol
    public static Bipoint[,] SendNewWaveHor(int height, int width, float distMax,
float angleRange, float dH, float hOffset, Vector3 position, Vector3 rotation, Color
color)
    {
        Bipoint[,] Data = Quadrillage.CreateEmptyMatrix(Bipoint.zero, height, width);
        float originalAngle = rotation.y - angleRange / 2f;
        float horAngle = angleRange / (width - 1);

        //Dans le sens de la hauteur
        for (int i = 0; i < height; i++)
        {
            Vector3 origine = position + new Vector3(0, hOffset + i * dH, 0);

            //Dans le sens de la largeur
            for (int j = 0; j < width; j++)
            {
                //On calcule l'angle horizontal
                float angle = j * horAngle + originalAngle;
                angle *= Mathf.Deg2Rad;
                Vector3 direction = new Vector3(Mathf.Sin(angle), 0f,
Mathf.Cos(angle));

                //On envoie un rayon et on regarde le rayon résultant
                Bipoint ray = new Bipoint(origine, direction * distMax, false);
                ray = SendRay(ray);
                Debug.DrawRay(ray.origine, ray.direction, color, Time.deltaTime);

                //On ajoute la distance obtenue à Data
                Data[i, j] = ray;
            }
        }
        return Data;
    }

    //Envoie une vague de rayons de façon cônique
    public static Bipoint[,] SendNewWaveCone(int height, int width, float distMax,
float horAngleRange, float vertAngleRange, float hOffset, Vector3 position, Vector3
rotation, Color color)
    {
        Bipoint[,] Data = Quadrillage.CreateEmptyMatrix(Bipoint.zero, height, width);
        float originalHAngle = rotation.y - horAngleRange / 2f;
        float originalVAngle = -rotation.x - vertAngleRange / 2f + 20;
        float horAngle = horAngleRange / (width - 1);
        float vertAngle = vertAngleRange / (height - 1);

        Vector3 origine = position + new Vector3(0, hOffset, 0);

        //Dans le sens de la hauteur
        for (int i = 0; i < height; i++)
        {
            float vAngle = i * vertAngle + originalVAngle;
```

```
            vAngle *= Mathf.Deg2Rad;
            //Dans le sens de la largeur
            for (int j = 0; j < width; j++)
            {
                //On calcule l'angle horizontal
                float hAngle = j * horAngle + originalHAngle;
                hAngle *= Mathf.Deg2Rad;
                Vector3 direction = new Vector3(Mathf.Sin(hAngle), Mathf.Sin(vAngle),
Mathf.Cos(hAngle));

                //On envoie un rayon et on regarde le rayon résultant
                Bipoint ray = new Bipoint(origine, direction * distMax, false);
                ray = SendRay(ray);

                if ((i == 0 && j == 0) || (i == height - 1 && j == 0) || (i == 0 && j
== width - 1) || (i == height - 1 && j == width - 1))
                {
                    Debug.DrawRay(ray.origine, ray.direction, color, Time.deltaTime);
                }

                //On ajoute la distance obtenue à Data
                Data[i, j] = ray;
            }
        }
        return Data;
    }


    //Calcule le trajet de ray en prennant en compte les colliders
    public static Bipoint SendRay(Bipoint ray)
    {
        float distMax = ray.magnitude;

        RaycastHit[] hitList;
        hitList = Physics.RaycastAll(ray.origine, ray.direction, distMax);

        RaycastHit hitMin = new RaycastHit { distance = distMax };

        foreach (RaycastHit hit in hitList)
        {
            if (hit.collider.gameObject.CompareTag("Obstacle") && hit.distance <
hitMin.distance)
            {
                hitMin = hit;
            }
        }

        if (hitMin.distance < distMax)
        {
            ray.flèche = hitMin.point;
        }

        return ray;
    }


    public static float[,] CreateDepthTable(Bipoint[,] Data)
    {
        int height = Data.GetLength(0);
        int width = Data.GetLength(1);

        float[,] depthTable = new float[height, width];
```

```
        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                depthTable[i, j] = Data[i, j].magnitude;
            }
        }
        return depthTable;
    }


    public static Texture2D EncodeDepthMap(float[,] depthTable, float distMax)
    {
        int height = depthTable.GetLength(0);
        int width = depthTable.GetLength(1);

        Texture2D depthMap = new Texture2D(width, height, TextureFormat.RGB24, false);

        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                float color = 1 - depthTable[i, j] / distMax;
                depthMap.SetPixel(j, i, new Color(color, color, color));
            }
        }
        depthMap.Apply();
        return depthMap;
    }
}
```

ObjectDetector

(esquisse d’un système de repérage des formes repérées par le lidar)

Obstacle

(structure pour repérer les formes)

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

public enum Shape
{
    Nsp,
    Cube,
    Rectangle,
    Boule,
    Cylindre,
}

public struct Obstacle
{
    public Vector2Int centerPosition;
    public float size;
    public Shape forme;

    public Obstacle(Vector2Int position, float size)
    {
        this.centerPosition = position;
        this.size = size;
        this.forme = Shape.Nsp;
    }

    public Obstacle(Vector2Int position, float size, Shape forme)
    {
        this.centerPosition = position;
        this.size = size;
        this.forme = forme;
    }
}


public class ObjectDetector : MonoBehaviour
{
    public PingManager pingManager;
    public GraphDisplay graphDisplay;

    public Vector2Int depthMapSize;
    public float viewDistance;
    public float hAngleRange;
    public float vAngleRange;
    public float hOffset;

    public Bipoint[,] Data;
    public float[,] depthTable;
    public List<Obstacle> obstacles;

    public bool render;

    public Color[] colors;
    public float deltaDist;
```

```
    public KeyCode takeColorMap;
    public bool save;


    void Update()
    {
        Data = Lidar.SendNewWaveCone(depthMapSize.x, depthMapSize.y, viewDistance,
hAngleRange, vAngleRange, hOffset,
            pingManager.transform.position,
pingManager.transform.rotation.eulerAngles, Color.gray);
        depthTable = Lidar.CreateDepthTable(Data);

        if (render)
        {
            graphDisplay.UpdateDepthMap(depthTable, viewDistance);
        }

        if (Input.GetKeyDown(takeColorMap))
        {
            FindObstacles(depthTable);
        }
    }

    private List<Vector2Int> FindNeighbouring(float[,] depthTable, Vector2Int pixel,
float deltaDist)
    {
        int height = Data.GetLength(0);
        int width = Data.GetLength(1);

        float dist = depthTable[pixel.x, pixel.y];

        List<Vector2Int> neighbours = new List<Vector2Int>();

        if (pixel.x > 0 && Mathf.Abs(depthTable[pixel.x - 1, pixel.y] - dist) <
deltaDist)
        {
            neighbours.Add(new Vector2Int(pixel.x - 1, pixel.y));
        }
        if (pixel.x < height - 1 && Mathf.Abs(depthTable[pixel.x + 1, pixel.y] - dist)
< deltaDist)
        {
            neighbours.Add(new Vector2Int(pixel.x + 1, pixel.y));
        }
        if (pixel.y > 0 && Mathf.Abs(depthTable[pixel.x, pixel.y - 1] - dist) <
deltaDist)
        {
            neighbours.Add(new Vector2Int(pixel.x, pixel.y - 1));
        }
        if (pixel.y < width - 1 && Mathf.Abs(depthTable[pixel.x, pixel.y + 1] - dist)
< deltaDist)
        {
            neighbours.Add(new Vector2Int(pixel.x, pixel.y + 1));
        }
        return neighbours;
    }


    //A partir d'un pixel, trouve tous ces voisins de sa composante connexe
    private List<Vector2Int> FindConnex(float[,] depthTable, int[,] coloration, int
shapeColor, Vector2Int originalPixel, float deltaDist)
    {
        if (coloration[originalPixel.x, originalPixel.y] != 0)
```

```
        {
            throw new System.ArgumentException("This pixel is already colored");
        }

        List<Vector2Int> neighbours = new List<Vector2Int> { originalPixel };
        coloration[originalPixel.x, originalPixel.y] = shapeColor;

        int next = 0;

        //On regarde tous les voisins trouvés dans la composante connexe, on s'arrête
quand il n'y en a plus
        while (next < neighbours.Count)
        {
            //On regarde les voisins du prochain pixel
            foreach (Vector2Int pixel in FindNeighbouring(depthTable,
neighbours[next], deltaDist))
            {
                if (coloration[pixel.x, pixel.y] == 0)
                {
                    coloration[pixel.x, pixel.y] = shapeColor;
                    neighbours.Add(pixel);
                }
                else if (coloration[pixel.x, pixel.y] < shapeColor)
                {
                    throw new System.ArgumentException("Case déjà colorée");
                }
            }
            next++;
        }
        return neighbours;
    }


    public void FindObstacles(float[,] depthTable)
    {
        int height = depthTable.GetLength(0);
        int width = depthTable.GetLength(1);

        int[,] coloration = new int[height, width];
        int c = 2;

        for (int i = 0; i < height; i++)
        {
            for (int j = 0; j < width; j++)
            {
                if (depthTable[i, j] >= viewDistance - deltaDist)
                {
                    coloration[i, j] = 1;
                }

                else if (coloration[i, j] == 0)
                {
                    FindConnex(depthTable, coloration, c, new Vector2Int(i, j),
deltaDist);
                    c++;
                    Debug.Log(c);
                }
            }
        }
        ColorMapToPNG(coloration);
    }
```

```
    private void ColorMapToPNG(int[,] coloration)
    {
        int height = coloration.GetLength(0);
        int width = coloration.GetLength(1);

        Texture2D texture = new Texture2D(width, height, TextureFormat.RGB24, true) {
filterMode = FilterMode.Point };

        for (int i = 0; i < texture.height; i++)
        {
            for (int j = 0; j < texture.width; j++)
            {
                try
                {
                    texture.SetPixel(j, i, colors[coloration[i, j] - 1]);
                }
                catch (System.IndexOutOfRangeException) { Debug.Log("Not enough
colors"); }
            }
        }

        texture.Apply(false);

        if (save)
        {
            byte[] image = texture.EncodeToPNG();
            //System.IO.File.WriteAllBytes("ColorMap.png", image);

            Debug.Log("The ColorMap has been successfully created");
        }
        else
        {
            StartCoroutine(graphDisplay.DispColorMap(texture));
        }
    }
}
```

OccupancyMap

(s’occupe de lire les données du Lidar pour les mettre dans des matrices correspondant à des couches de cartes)

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

//Les différents types de case possibles
public enum MCode
{
    Nsp = 0,
    Vide = 1,
    Surface = 2,
}

public class OccupancyMap : MonoBehaviour
{
    public Vector3Int size;

    public Vector3 limitUp;
    public Vector3 limitDown;

    public Quadrillage quadrillage;

    public MCode[,,] carte;

#pragma warning disable IDE0051
    void Start()
    {
        quadrillage = new Quadrillage
            (size.z, size.y, size.x, limitUp.z - limitDown.z, limitUp.y - limitDown.y,
limitUp.x - limitDown.x, false);

        carte = Quadrillage.CreateEmptyMatrix3<MCode>(MCode.Nsp, size.z + 1, size.x +
1, size.y);
    }
#pragma warning restore IDE0051

    //Lit les trajectoires reçues pour les mettre sur la carte
    public void UpdateMap(Bipoint[,] Data)
    {
        //On ajoute chaque parcours dans une liste
        foreach (Bipoint ray in Data)
        {
            List<Vector3Int> parcours = quadrillage.Parcours(DansQuadrillage(ray));

            try
            {
                foreach (Vector3Int place in parcours)
                {
                    carte[place.x, place.y, place.z] = MCode.Vide;
                }
            }
            catch (ArgumentOutOfRangeException) { }
            catch (IndexOutOfRangeException) { }
            finally { }
        }
    }
```

```
    //Transforme un Vector3 centré en 0 en Vector3 centré au début de Quadrillage.
    public Vector3 DansQuadrillage(Vector3 vector3)
    {
        return vector3 - new Vector3(limitDown.x, limitDown.y, limitUp.z);
    }

    //Transforme un Bipoint centré en 0 en Bipoint centré au début de Quadrillage.
    public Bipoint DansQuadrillage(Bipoint bipoint)
    {
        return new Bipoint(DansQuadrillage(bipoint.origine),
DansQuadrillage(bipoint.flèche));
    }
}
```

PingManager

(envoye des rayon par le Lidar pour envoyer les données à OccupancyMap)

RayInfo

(structure de rayon envoyé par le Lidar)

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

public struct RayInfo
{
    public Bipoint ray;
    public bool touched;

    public RayInfo(Bipoint ray, bool touched)
    {
        this.ray = ray;
        this.touched = touched;
    }
}


public class PingManager : MonoBehaviour
{
    public GameObject simManager;
    private OccupancyMap occupancyMap;

    public int nbHor;                                  //nbHor correspond au nombre
de rayons envoyés sur le plan (xOz)
    public int nbVert;                                 //nbVer correspond au nombre
de rayons envoyés selon l'axe (Oy)
    public float hOffset;
    public float dH;
    public float angleRange;

    public float distMax;

    public Bipoint[,] Data;
    public List<RayInfo[,]> pingTable = new List<RayInfo[,]>();

    private Vector3 lastPos;
    private Quaternion lastRot;

#pragma warning disable IDE0051
    void Start()
    {
        occupancyMap = simManager.gameObject.GetComponent<OccupancyMap>();
        lastPos = transform.position;
        lastRot = transform.rotation;

        Data = Lidar.SendNewWaveHor
                (nbVert, nbHor, distMax, angleRange, dH, hOffset, transform.position,
transform.rotation.eulerAngles, Color.green);
    }

    void Update()
    {
        if (lastPos != transform.position || lastRot != transform.rotation)
        {
```

```
            Data = Lidar.SendNewWaveHor
                (nbVert, nbHor, distMax, angleRange, dH, hOffset, transform.position,
transform.rotation.eulerAngles, Color.green);
            occupancyMap.UpdateMap(Data);

            lastPos = transform.position;
            lastRot = transform.rotation;
        }
    }
#pragma warning restore IDE0051
}
```

Quadrillage

(classe gérant la position des indices dans une matrice ainsi que la correspondance matrice – monde)

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

public class Quadrillage
{
    public int zAxisNb;
    public int yAxisNb;
    public int xAxisNb;

    public float zScale = 1;
    public float yScale = 1;
    public float xScale = 1;

    //Crée un quadrillage vide
    public Quadrillage() { }

    //Crée un quadrillage en choisissant la taille
    public Quadrillage(int zAxisNb, int yAxisNb, int xAxisNb)
    {
        this.zAxisNb = zAxisNb;
        this.yAxisNb = yAxisNb;
        this.xAxisNb = xAxisNb;
    }

    //Crée un quadrillage complet
    public Quadrillage(int zAxisNb, int yAxisNb, int xAxisNb, float longueur, float
hauteur, float largeur, bool useScalesInstead = false)
    {
        this.zAxisNb = zAxisNb;
        this.yAxisNb = yAxisNb;
        this.xAxisNb = xAxisNb;

        //Méthode avec les distances
        if (!useScalesInstead)
        {
            zScale = Mathf.Abs(longueur / zAxisNb);
            yScale = Mathf.Abs(hauteur / yAxisNb);
            xScale = Mathf.Abs(largeur / xAxisNb);
        }
        //Méthode avec les divisions
        else
        {
            zScale = Mathf.Abs(longueur);
            yScale = Mathf.Abs(hauteur);
            xScale = Mathf.Abs(largeur);
        }
    }

#pragma warning disable IDE1006
    //Donne les paramètres de taille
    public float longueur
    {
        get => zScale * zAxisNb;
        set => zScale = Mathf.Abs(longueur / zAxisNb);
    }

    public float hauteur
    {
```

```
        get => zScale * zAxisNb;
        set => zScale = Mathf.Abs(hauteur / zAxisNb);
    }

    public float largeur
    {
        get => xScale * xAxisNb;
        set => xScale = Mathf.Abs(largeur / zAxisNb);
    }
#pragma warning disable IDE1006

    //Transforme un Vector3 float en Vector3Int adapté au quadrillage. (i : axe -z, j
: axe +x, k : axe +y)
    public Vector3Int Point(Vector3 vector)
    {
        int i = -vector.z != longueur ? (int)(-vector.z / zScale) : (int)(-vector.z /
zScale) - 1;

        int j = vector.x != largeur ? (int)(vector.x / xScale) : (int)(vector.x - 1 /
xScale) - 1;

        int k = (int)(vector.y / yScale);

        return new Vector3Int(i, j, k);
    }


    //Transpose dans le quadrillage tous les points que rencontre le bipoint
(seulement sur le plan (x,z))
    public List<Vector3Int> Parcours(Bipoint bipoint)
    {
        Vector3Int origine = Point(bipoint.origine);
        Vector3Int flèche = Point(bipoint.flèche);

        int a = Mathf.Abs(flèche.x - origine.x);
        int b = Mathf.Abs(flèche.y - origine.y);

        int n = (int)Mathf.Sqrt(a * a + b * b);

        List<Vector3Int> parcours = new List<Vector3Int>();

        //Si la trajectoire n'est pas réduite à un point
        if (n != 0)
        {
            parcours.Add(Point(bipoint.origine + (1 / n) * bipoint.direction));

            //Sépare le bipoint en Bipoints plus courts
            for (float k = 2; k < n + 1; k++)
            {
                Vector3Int coord = Point(bipoint.origine + (k / n) *
bipoint.direction);
                if (coord != parcours[parcours.Count - 1])
                {
                    parcours.Add(coord);
                }
            }
        }
        return parcours;
    }

    //Crée une matrice cubique de type T et de taille (n,p,q) de obj objets
    public static T[,,] CreateEmptyMatrix3<T>(T obj, int n, int p, int q)
```

```
    {
        T[,,] arr = new T[n, p, q];

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < p; j++)
            {
                for (int k = 0; k < q; k++)
                {
                    arr[i, j, k] = obj;
                }
            }
        }
        return arr;
    }

    //Crée une matrice de type T et de taille (n,p) de obj objets
    public static T[,] CreateEmptyMatrix<T>(T obj, int n, int p)
    {
        T[,] arr = new T[n, p];

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < p; j++)
            {
                arr[i, j] = obj;
            }
        }
        return arr;
    }

    //Crée un vecteur de type T et de taille n de obj objets
    public static T[] CreateEmptyArray<T>(T obj, int n)
    {
        T[] arr = new T[n];

        for (int i = 0; i < n; i++)
        {
            arr[i] = obj;
        }
        return arr;
    }
}
```

RobotController

(gère les entrées utilisateur et fait bouger le robot)

```
using UnityEngine;

public enum Side
{
    Left = 1,
    Right = -1,
}

public class RobotController : MonoBehaviour
{
    private Rigidbody rbRigidbody;
    public GameObject centerOfMass;
    public CaterpillarMover lCaterpillar;
    public CaterpillarMover rCaterpillar;

    public float inputX;
    public float inputY;

    public float powerMax;
    public float turnRate;
    public float rearRate;

#pragma warning disable IDE0051
    private void Start()
    {
        rbRigidbody = gameObject.GetComponent<Rigidbody>();

        rbRigidbody.centerOfMass = centerOfMass.transform.localPosition;
    }

    void Update()
    {
        inputX = Input.GetAxis("Horizontal");
        inputY = Input.GetAxis("Vertical");

        if (inputX != 0 || inputY != 0)
        {
            rbRigidbody.WakeUp();
        }
    }
#pragma warning disable IDE0051

    private void OnCollisionStay(Collision collision)
    {
        //Pour chaque point de contact, appliquer une force au niveau du point
d'application normalement à la surface du collider
        foreach (ContactPoint contact in collision.contacts)
        {
            lCaterpillar.Move(contact, Side.Left, Color.blue);
            rCaterpillar.Move(contact, Side.Right, Color.red);
        }
    }
}
```

TitleScreen

(gère l’interface utilisateur)

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

public class TitleScreen : MonoBehaviour
{
    public List<GameObject> textsList;
    public float secondsBeforeDisplay;
    private bool called = false;

    void Update()
    {
        if (Input.anyKey)
        {
            called = false;
            foreach (var text in textsList)
            {
                text.SetActive(false);
            }
        }
        else if (!called)
        {
            StartCoroutine(Show());
        }
    }

    private IEnumerator Show()
    {
        called = true;
        float time = Time.time;
        while(!Input.anyKey && Time.time < time + secondsBeforeDisplay)
        {
            yield return new WaitForEndOfFrame();
        }
        if (!Input.anyKey)
        {
            foreach (var text in textsList)
            {
                text.SetActive(true);
            }
        }
        called = false;
    }

    public void Close()
    {
        Application.Quit();
    }
}
```