Dynamic Path Generation System for Continuous Walking in Finite Space using PragPal algorithm in VR

Kandarp Devmurari

1 Problem Statement

Explore a large virtual environment in a limited physical space (without external hardware support like an omnidirectional treadmill) by developing an algorithm that dynamically generates wall boundaries (for the user to move through in VR) on the basis of how far/close the user is to the finite room boundary.

2 My contributions

1. Tried to visualize and implement an alternate way to generate parallel wall boundaries for PragPal algorithm.
2. Made a 3D endless runner game in Unity and then converted it into a VR game which is played using the oculus controller.
3. Imported the assets from the 3D game and integrated it with the PragPal algorithm.

3 Detailed Report

3.1 New approach to visualize parallel path boundaries in PragPal algorithm

Introduction

Visualizing a new robust method to generate boundaries around the dynamic path made using PragPal algorithm

Previous Method

The previous method was based on a simple approach in which we kept the path width constant except at the turns. In this method the path width at turn,

$$p_{tw} = p_w / sin(\theta/2) \tag{1}$$

where $\theta = 180°$ - β, β is the turn angle between the 2 path segment vectors.

Issue with the above method

The above method causes distortion in boundary walls when new walls are added. Since the path width is constant, the path width at the end point is p_w(as there is no more path segment for β to exist). But when a new path segment is added at this end point, the path width at this point changes from p_w to p_{tw}, because of this the previous walls distort.

New Method

Using this method we can generate distortion-less parallel path boundaries but with non-constant width.

Procedure and math involved

Let us assume a 2-turn path using 4 points P_1, P_2, P_3 and P_4. Let P_1 is the start point and P_4 is the end point. Let the starting path width be p_w, $P_2 - P_1 = \overrightarrow{a}$, $P_3 - P_2 = \overrightarrow{b}$ and turn angle between $\overrightarrow{a}$ and $\overrightarrow{b}$ be β. Now we try to find the direction of average of the 2 vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ to find the path boundary points at the turn.(see Figure 1)

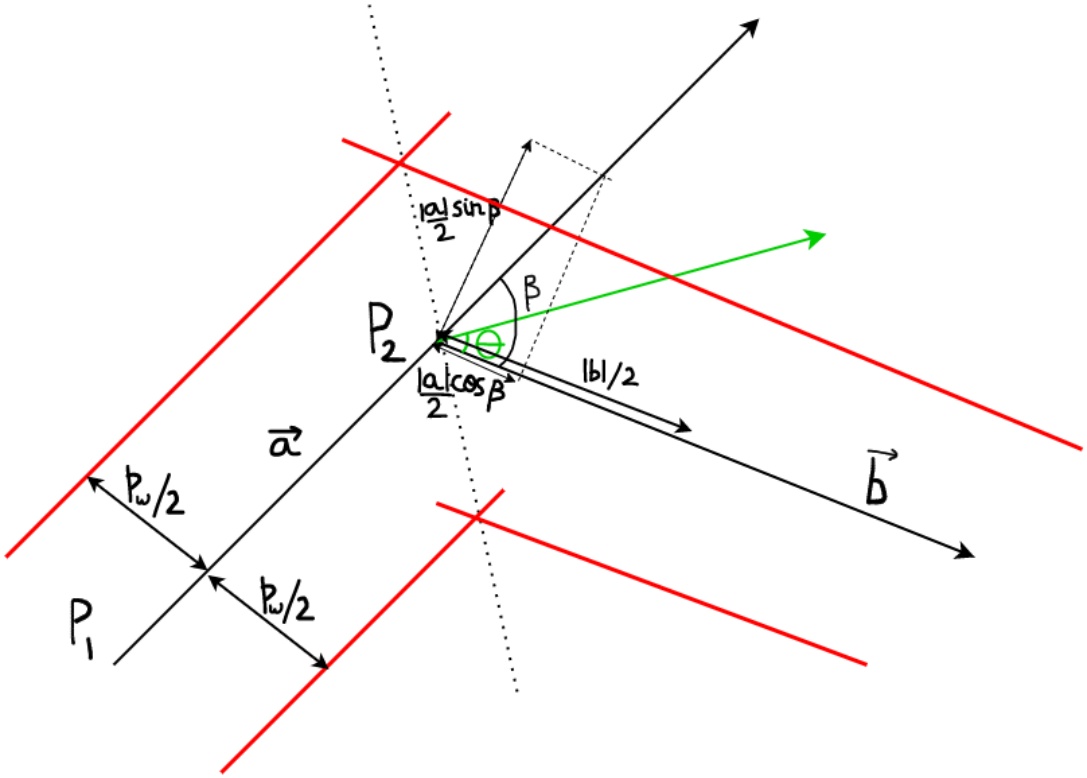

Figure 1: Finding boundary points and path width at a turn

By taking components of vector $\overrightarrow{a}$ perpendicular and parallel to $\overrightarrow{b}$ we get the average direction of the vectors. The average direction vector has a angle θ with the $\overrightarrow{b}$. Once we get the average vector direction we draw a line perpendicular to this vector at the turn point. Our boundary points will lie on this line.

By calculations from the Figure 1 and Figure 2 we get the values of θ(the angle of the average vector of $\overrightarrow{a}$ and $\overrightarrow{b}$ with $\overrightarrow{b}$), p_{tw}(path width at turn) and p_{pw}path width of path after the turn.

$$\theta = tan^{-1}\left(\frac{|a|sin(\beta)}{|b| + |a|cos(\beta)}\right) \tag{2}$$

$$p_t w = 2x = p_w \cdot sec(\beta - \theta) = p_w \cdot sec\left(\beta - tan^{-1}\left(\frac{|a|sin(\beta)}{|b| + |a|cos(\beta)}\right)\right) \tag{3}$$

$$p_p w = 2xcos(\theta) = p_t w \cdot cos(\theta) = p_w \cdot \left(\frac{cos(\theta)}{cos(\beta - \theta)}\right) = p_w \cdot \left(\frac{|b| + |a|cos(\beta)}{|a| + |b|cos(\beta)}\right) \tag{4}$$

NOTE: For $p_{pw} > p_w$, we need

$$\left(\frac{|b| + |a|cos(\beta)}{|a| + |b|cos(\beta)}\right) > 1 \tag{5}$$

$$|b| + |a|cos(\beta) > |a| + |b|cos(\beta) \tag{6}$$

$$|b| - |a| > (|b| - |a|)cos(\beta) \tag{7}$$

$$(|b| - |a|)(1 - cos(\beta)) > 0 \tag{8}$$

In the above equation, $1 - cos(\beta)$ will always be greater than 0 (unless β is 0, which implies $p_{pw} = p_w$), so for $p_{pw} > p_w$, |b| should be greater than |a| (if |b| = |a|, then again $p_{pw} = p_w$), which is always true for our algorithm. Therefore, in my method, the path width does not shrink after a turn has been made, it may increase or remain the same.

We do the same as above for all the turn points and find the line on which boundary points will lie. Then from the starting point P_1 we draw 2 parallel lines to the $\overrightarrow{a}$ and let it intersect the boundary line made previously, these intersection points will give us the left and right boundary points.

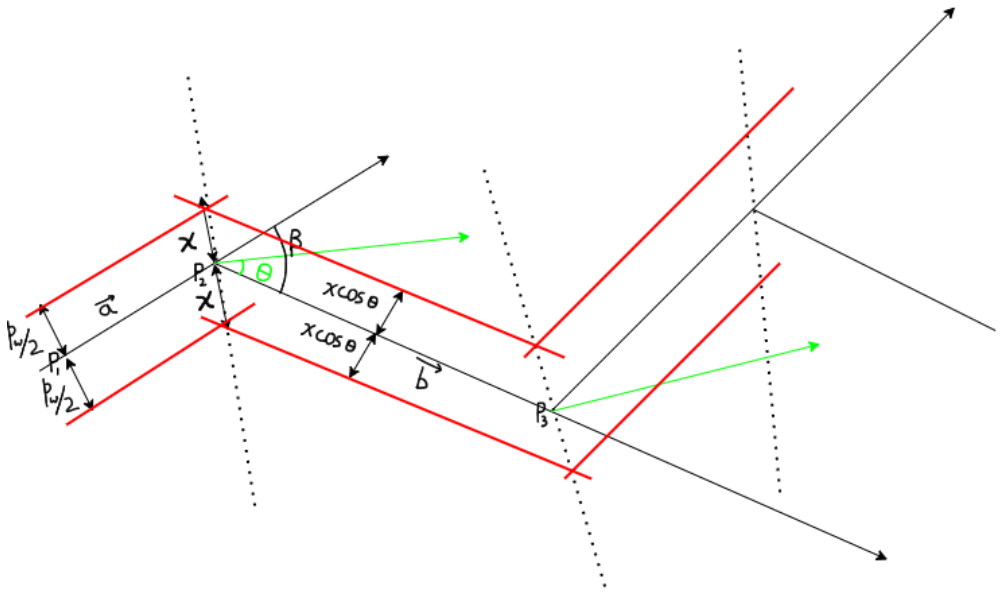

Figure 2: Generation of parallel path with non-constant path width

We repeat the above by extrapolating a parallel path again from the above newly found boundary points along the $\overrightarrow{b}$ and let it intersect the next boundary line. Using this we can generate a parallel path with non-constant width.

However, some distortion may occur at the endpoint if we are not calculating the next iterations path vector(initially the end point width was some $p_w^{'}$ but after a new path is added in front of it the path boundary points change making the turn path width a little longer and skewed from the original path width). This problem can be avoided if we calculate the path vector of the next segment before rendering it in VR.

Certain constraints for proper functioning of above method

1. For acute β,
   (a) There is NO wall intersection problem for |b| $\geq$ |a| or |b| $<$ |a|.
   (b) For |b| $\geq$ |a|, the path expands (i.e $p_{pw} \geq p_w$), $p_{pw} \in [p_w, p_w/cos\beta)$.
   (c) For |b| $<$ |a|, the path shrinks (i.e $p_{pw} \leq p_w$), $p_{pw} \in (p_w cos\beta, p_w]$.
2. For obtuse β,
   (a) There is NO wall intersection problem for $|b| < |acos\beta|$ but the problem arises when $|b| \geq |acos\beta|$.
   (b) Condition for p_{pw} to exist: $|b|/|a| > -cos\beta$ & $|a|/|b| > -cos\beta$, i.e $|cos\beta|$ should be less than both $|b|/|a|$ & $|a|/|b|$.
   (c) $|b| > |a|$ works but $|a| > |b|$ doesn't work.
   (d) **Thumb rule: For obtuse β always keep the next segment path length(|b|) such that $|a|/cos\beta > |b| > |a|$, for proper walls**. Also, note that since |b| $>$ |a|, the path width always expands for obtuse β.

Issue solved by above method

In this method if we know the (unrendered)path segment vector after the end point, so we can get boundary walls with no/minimal distortion/narrowing. Also, even if we get some distortion if will not affect the previous wall boundaries.

Code changes implemented in PragPal algorithm in place of previous method

```
else{
    a = points[i] - points[i-1];
    b = points[i+1] - points[i];
    float a_mag = Vector3.Magnitude(a);
    float b_mag = Vector3.Magnitude(b);
    float calc_beta = Vector3.Angle(a,b);

    float val_to_arcTan =
    (a_mag * Mathf.Sin(Mathf.Deg2Rad * (calc_beta)))/(b_mag + a_mag * Mathf.Cos(Mathf.Deg2Rad * (calc_beta)));

    pathWidthMultiplier = 1/Mathf.Cos(Mathf.Deg2Rad * ((calc_beta) - Mathf.Rad2Deg * Mathf.Atan(val_to_arcTan)));
}
pathWidth = pathWidth * pathWidthMultiplier;
```

Figure 3: Code change to calculate pathWidthMultiplier using new method in GenerateLeftRight-Points() function

```
float dynamicPathLength;
dynamicPathLength = pathLength;
//change by Kandarp
float previousLength = Vector3.Magnitude(lastPoint - pointsList[pointsList.Count - 2]);
if(Mathf.Abs(newBeta) % Mathf.PI > Mathf.PI/2)
{
    if(!(dynamicPathLength > previousLength && dynamicPathLength < previousLength/Mathf.Cos(newBeta))){
        dynamicPathLength = UnityEngine.Random.Range(previousLength,previousLength/Mathf.Cos(newBeta));
    }
}
```

Figure 4: Code change to apply the constraints so that the walls are generated properly even for sharp turns

3.2 3D endless runner game in Unity

Introduction

The game was made in 3D first and afterwards integrated with PragPal algorithm to work in VR also; as it would be easier to do this instead of directly making changes in the PragPal project(which could lead to more time consumption and more errors)
I took inspiration for the game's overall color scheme(of bright orange and blue), the wall design and the character from the movie Tron Legacy.

Game components

1. **Walls:** The game has 2 types of walls - orange and blue.

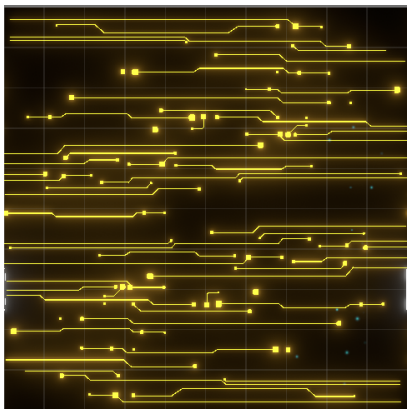

Figure 5: Orange Wall

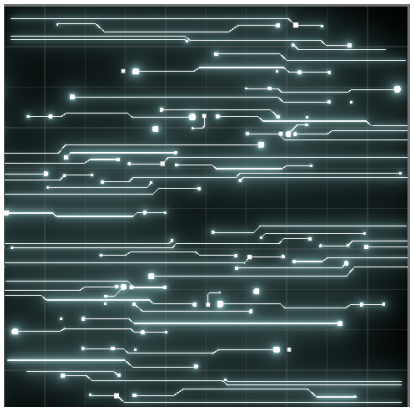

Figure 6: Blue Wall

2. **Obstacles:** The game has 3 types of obstacles/fields - a full plane, a duck and a jump obstacle(all 3 purple colored). If we touch or try to pass these obstacles the GAME OVERS.

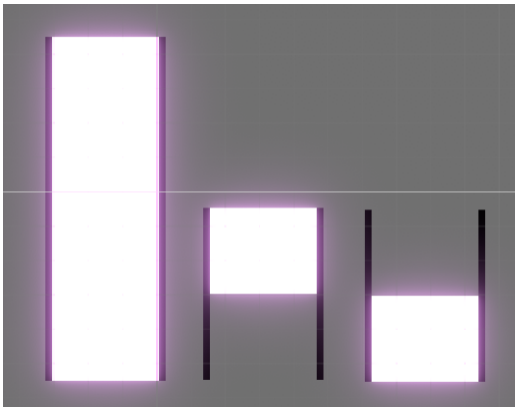

Figure 7: Obstacles

3. **Gates:** The game has 2 types of gates - orange and blue. The player has to pass the gate by changing its color to the color of the gate, else the GAME OVERS.

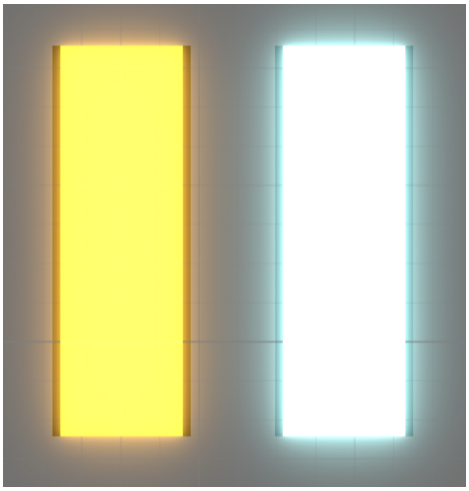

Figure 8: gates

4. **Turrets, bullets and discs:** The game has 2 types of turrets, bullets and discs - orange and blue. The player can destroy the turrets by changing its color to the color opposite of the turret's color and hitting it with the disc. The bullets hitting the player decreases the player's health.

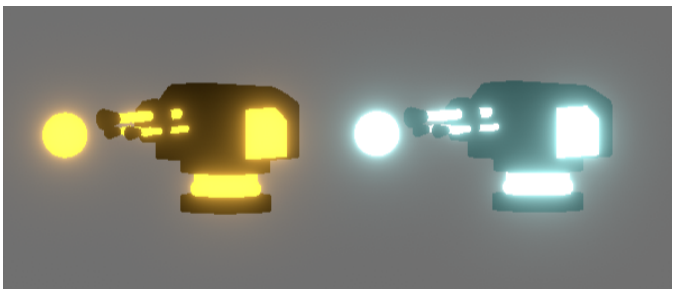

Figure 9: Turrets and bullets

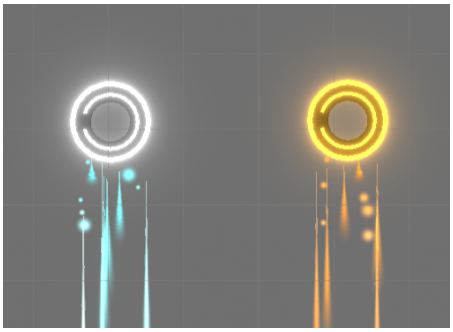

Figure 10: Discs

5. **Player character:** The game has 2 types of player - orange and blue. The player's color can be toggled between orange and blue to perform certain tasks(like passing a particular color gate or destroying a opposite color turret).

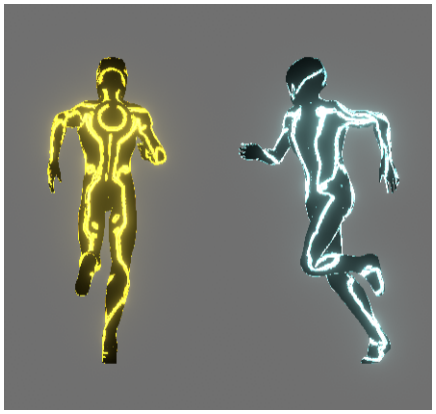

Figure 11: Player character

6. **Coin**: In game we can collect coins to increase our score.

7. **Audio Manger**: This game object has 3 sounds - Main theme, coin pickup and GAME OVER.

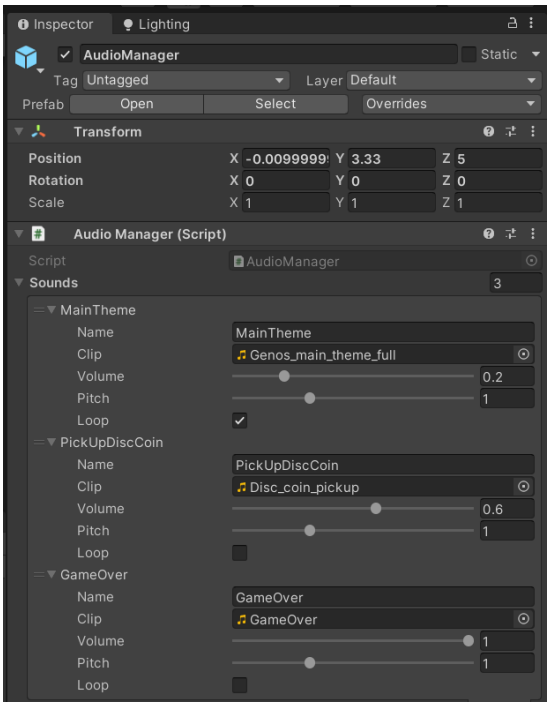

Figure 12: Audio Manager

Game keys and mechanics

1. Right and left keyboard keys to move right and left, up and down keyboard keys to jump and duck respectively.
2. Space bar key to toggle the color of the player, walls, disc and the background between blue and orange.
3. Right mouse click to start the game and left mouse click to shoot discs.
4. The obstacles, gates and turrets of are generated randomly(in type and amount), so the user doesn't feel that the game repeats and is thus boring.
5. The player receives the final score on the basis of coins collected, turrets destroyed and gates passed.
6. The player's speed keeps on increasing in order to increase the difficulty

3.3 3D endless runner game converted to a VR game

Introduction

Here, I converted the above 3D endless runner game to work in Oculus VR.

Changes made to convert the game from 3D to VR

1. Several packages like XR Plug-in Management, XR interaction toolkit, oculus XR Plug-in and oculus integration had to be installed.
2. The canvas which supported the start, game over and score panel had to be changed from screen space-overlay to worldspace, and its scale had to be changed to match the view in oculus.
3. All the keyboard inputs had to be converted to oculus controller inputs using OVR Input script which is present in the oculus integration package.
4. OVRCameraRig was added to the player which tracks the movement of the controller and the user in oculus VR.
5. Since mouse can no longer be used to do onclick events, raycasters had to be the added to the controllers by adding the UIHelpers prefab.

NOTE: Right now, the VR game is played using controller's joysticks(i.e no actual movement of the player).But to convert it into a game which requires actual player movement; we can remove the PlayerController script and add a Locomotion system to the scene

3.4 Integrating the 3D game assets with PragPal algorithm

Introduction

Converted the 3D game assets into a unitypackage and imported it in the PragPal algorithm projects - demo_simulation(to test things in Unity editor) and TillDocIDInputPanel(the latest PragPal project to build and run the changes in VR)

Code implemented in PragPal algorithm to spawn obstacles

The code was added mainly in DetectBoundaryTestScript and WallSpawner scripts.

1. I wrote a GenerateObstacleGateTurretCoin() function in WallSpawner to generate obstacles, gates, turrets and coins along the walls generated by PragPal algorithm.

2. The code spawns the above mentioned objects on the wall which has longer length(this is done to prevent problems caused due to object spawning at small walls and sharp turns). This is done by below code segment.

```
float leftWallLength = (points[0][points[0].Count - 2] - points[0][points[0].Count - 3]).magnitude;
float rightWallLength = (points[1][points[1].Count - 2] - points[1][points[1].Count - 3]).magnitude;

float smallerLength = Mathf.Max(leftWallLength,rightWallLength);
bool isRightWall = false;

if(smallerLength == rightWallLength){
    isRightWall = true;
}
else{
    isRightWall = false;
}
```

Figure 13: Code to decide whether to spawn near right or left wall

3. The below code segment is used to generate the objects and their color type randomly.

```
int obstacleORgateORturretORcoin = Random.Range(0,4);
// 0 for obstacle, 1 for gate, 2 for turret, 3 for coin

int colorType = Random.Range(0,2); // 0 for blue, 1 for orange
GameObject obstacleORgateORturretORcoininstantiate;
if(obstacleORgateORturretORcoin == 0){
    obstacleORgateORturretORcoininstantiate = obstacleBlueGateOrangegate[0];
}
else if(obstacleORgateORturretORcoin == 1){
    if(colorType == 0){
        obstacleORgateORturretORcoininstantiate = obstacleBlueGateOrangegate[1];
    }
    else{
        obstacleORgateORturretORcoininstantiate = obstacleBlueGateOrangegate[2];
    }
}
else if(obstacleORgateORturretORcoin == 2){
    if(colorType == 0){
        obstacleORgateORturretORcoininstantiate = turretBlueOrange[0];
    }
    else{
        obstacleORgateORturretORcoininstantiate = turretBlueOrange[1];
    }
}
else{
    obstacleORgateORturretORcoininstantiate = coin;
}
```

Figure 14: Code to spawn objects randomly

4. The code spawns the objects only if the smaller wall is longer than a certain path limit and if the turn angle(β) is acute. This prevents the objects from spawning at sharp turns(where β is obtuse) and at very small wall lengths.This is done by below code's if condition.

```
float pathLimit = 0.25f;
if(smallerLength > pathLimit && (Mathf.Abs(betaList[betaList.Count - 2]) % Mathf.PI) <= Mathf.PI/3){
    GameObject tmp = Instantiate(obstacleORgateORturretORcoininstantiate, Vector3.zero, Quaternion.identity);
    Vector3 tmpScale = tmp.transform.localScale;
```

Figure 15: Code to limit spawning of object for a particular path length and β only

5. The code spawns the object whose scale depends on the path width for that iteration's walls, so that the objects fit properly between the 2 walls with changing path width.

```
Vector3 tmpScale = tmp.transform.localScale;
float pathWidth = (points[1][points[1].Count - 2] - points[0][points[0].Count - 2]).magnitude;
if(obstacleORgateORturretORcoin == 2){
    float turretScale = pathWidth/5;
    tmpScale.x = turretScale;
    tmpScale.y = turretScale;
    tmpScale.z = turretScale;
}
else if(obstacleORgateORturretORcoin == 3){
    float coinScale = pathWidth/3;
    tmpScale.x = tmp.transform.localScale.x * coinScale;
    tmpScale.y = tmp.transform.localScale.y * coinScale;
    tmpScale.z = tmp.transform.localScale.z * coinScale; // do for turret and walls also like
}
else{
    tmpScale.x = pathWidth/3; // here 1/3 factor can be changed
    tmpScale.y = 2f;
    tmpScale.z = 0.01f;
}

tmp.transform.localScale = tmpScale;
```

Figure 16: Code to scale the objects on the basis of path width

6. The code spawns the objects either on left lane or right lane. Left lane is the lane which is in the middle of the left wall and the middle line path, similarly for right lane. The middle line path point is calculated by taking average of 4 points(right points for i and i-1 iteration and left points for i and i-1 iteration, as averaging these gives the center point of the quadrilateral formed by these 4 points). Then the mid point between the center point and the mid point of the left/right wall(depending on where to spawn) is found to spawn the object at that position.

```
Vector3 tmpPos = tmp.transform.position;
Vector3 leftMid = (points[0][points[0].Count - 2] + points[0][points[0].Count - 3])/2;
Vector3 rightMid = (points[1][points[1].Count - 2] + points[1][points[1].Count - 3])/2;
Vector3 mid =
(points[1][points[1].Count - 2] + points[1][points[1].Count - 3]
+ points[0][points[0].Count - 2] + points[0][points[0].Count - 3])/4;
Vector3 spawnPoint;
if(isRightWall){
    spawnPoint = (mid + rightMid)/2;
}
else{
    spawnPoint = (mid + leftMid)/2;
}

tmpPos = spawnPoint;

if(obstacleORgateORturretORcoin == 2){
    tmpPos.y = tmpScale.y;
}
else if(obstacleORgateORturretORcoin == 3){
    tmpPos.y = 1.32f;
}
else{
    tmpPos.y = tmpScale.y - 0.8f;
}
tmp.transform.position = tmpPos;
```

Figure 17: Code to find the position of the object to spawn the object

7. This code spawns the object with an orientation of 90° with the respective wall length.

```
Vector3 leftWall = (points[0][points[0].Count - 2] - points[0][points[0].Count - 3]);
Vector3 rightWall = (points[1][points[1].Count - 2] - points[1][points[1].Count - 3]);
float obstacleAngleAlongY;
if(isRightWall){
    obstacleAngleAlongY =
    Vector3.SignedAngle(new Vector3(1f, 0f, 0f), Quaternion.AngleAxis(90,Vector3.up)*rightWall.normalized, Vector3.up);
}
else{
    obstacleAngleAlongY =
    Vector3.SignedAngle(new Vector3(1f, 0f, 0f), Quaternion.AngleAxis(90,Vector3.up)*leftWall.normalized, Vector3.up);
}

Vector3 rotation = tmp.transform.localEulerAngles;
rotation = new Vector3(0f, obstacleAngleAlongY, 0f);
tmp.transform.localEulerAngles = rotation;
```

Figure 18: Code to orient the wall 90° wrt the wall

4 Github repo and YT demo link

Github link: `https://github.com/VHIL-interns-hub/New-boundary-generation-PragPal.git`
YT demo links:

1. 3D version of the VR game: `https://youtu.be/wDynDDhyhts`
2. Simulation of obstacle generation in PragPal algorithm: `https://youtu.be/hTnje2lkwhs`