FISHING GAME TOOL

Fishing System Development Tool for Unity

ADVANCED GUIDE

Release 1.2

10.2023

Created by Paweł Mularczyk

FISHING GAME TOOL

Table of Contents

FISHING GAME TOOL

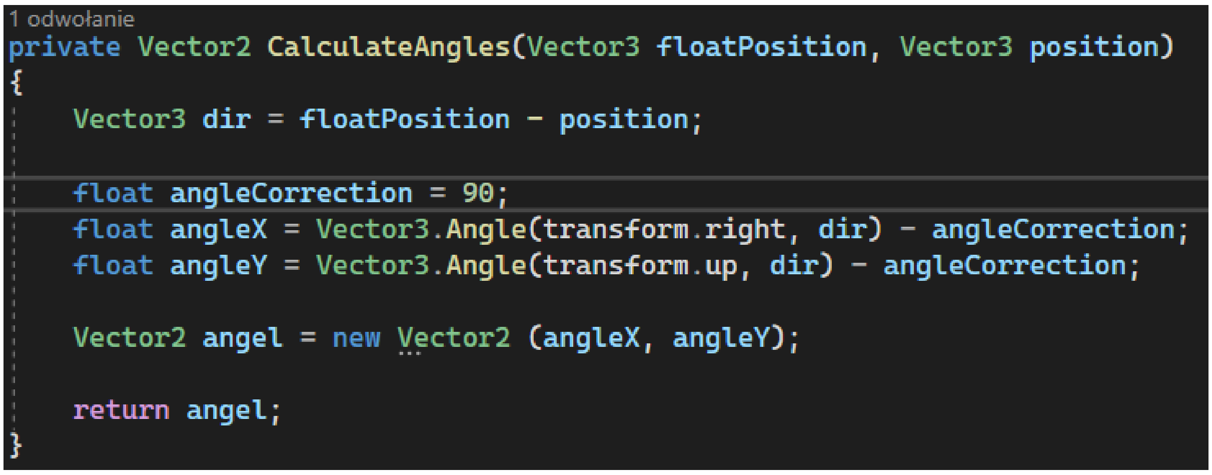
Overview

This document will guide you through the advanced configuration options of the fishing system. It will explain which elements in the tool's code are responsible for the various mechanisms of the system. This includes everything from speed calculations to the procedural behavior of a fish in a designated body of water.

FISHING GAME TOOL

Calculate Bend

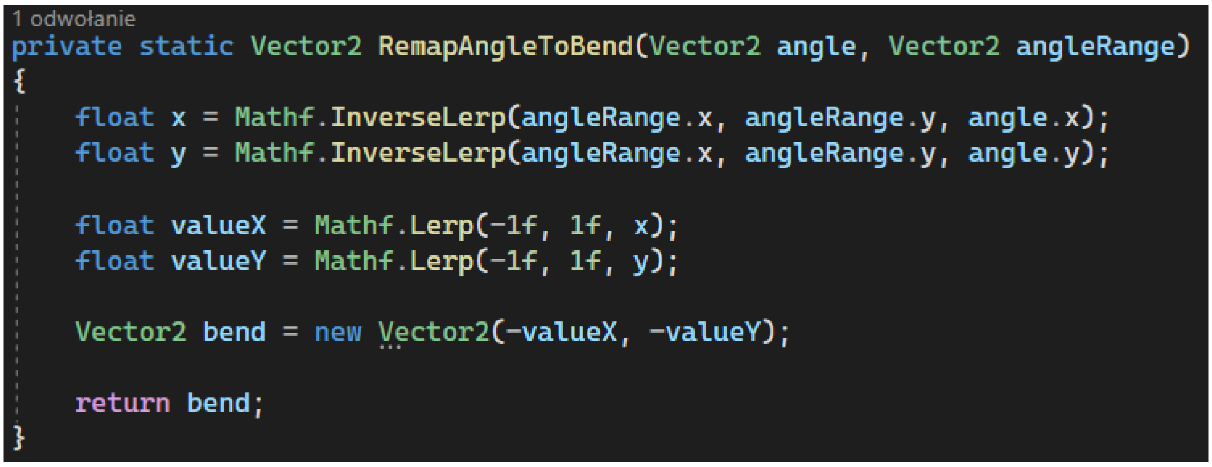
Rod bending is achieved through the utilization of an animator and a blend tree. Initially, the calculation involves determining the angles along the X and Y axes between the fishing rod and the float.



```
1 odwołanie
private Vector2 CalculateAngles(Vector3 floatPosition, Vector3 position)
{
    Vector3 dir = floatPosition - position;

    float angleCorrection = 90;
    float angleX = Vector3.Angle(transform.right, dir) - angleCorrection;
    float angleY = Vector3.Angle(transform.up, dir) - angleCorrection;

    Vector2 angel = new Vector2 (angleX, angleY);

    return angel;
}
```

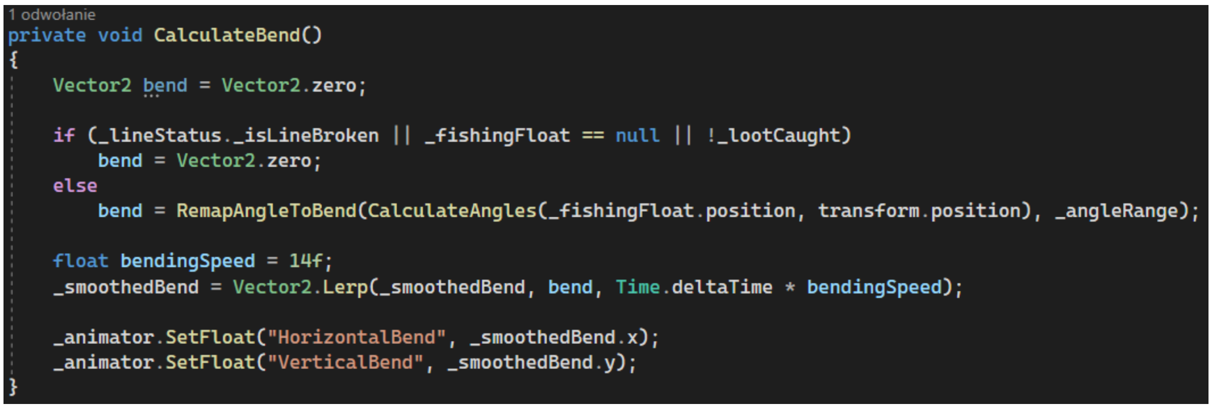
angleCorrection involves setting the angle for the x and y axes equal to 0 when the float is centrally in front of the rod.

Afterward, the resulting Vector2 is mapped to the values compatible with the animator's requirements, specifically ranging from -1 to 1 for both the x and y axes.



```
1 odwołanie
private static Vector2 RemapAngleToBend(Vector2 angle, Vector2 angleRange)
{
    float x = Mathf.InverseLerp(angleRange.x, angleRange.y, angle.x);
    float y = Mathf.InverseLerp(angleRange.x, angleRange.y, angle.y);

    float valueX = Mathf.Lerp(-1f, 1f, x);
    float valueY = Mathf.Lerp(-1f, 1f, y);

    Vector2 bend = new Vector2(-valueX, -valueY);

    return bend;
}
```

During the last stage, the Vector2 values undergo a smoothing process and are subsequently transmitted to the animator.

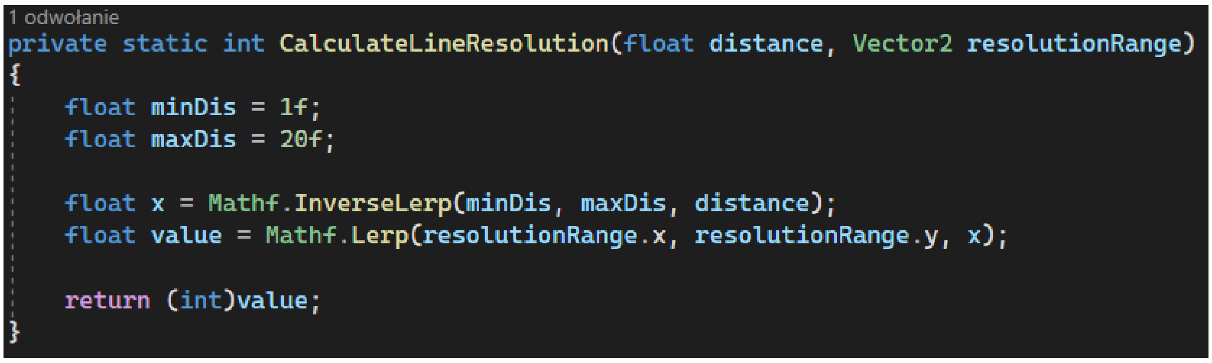


```
1 odwołanie
private void CalculateBend()
{
    Vector2 bend = Vector2.zero;

    if (_lineStatus._isLineBroken || _fishingFloat == null || !_lootCaught)
        bend = Vector2.zero;
    else
        bend = RemapAngleToBend(CalculateAngles(_fishingFloat.position, transform.position), _angleRange);

    float bendingSpeed = 14f;
    _smoothedBend = Vector2.Lerp(_smoothedBend, bend, Time.deltaTime * bendingSpeed);

    _animator.SetFloat("HorizontalBend", _smoothedBend.x);
    _animator.SetFloat("VerticalBend", _smoothedBend.y);
}
```

FISHING GAME TOOL

Fishing Line

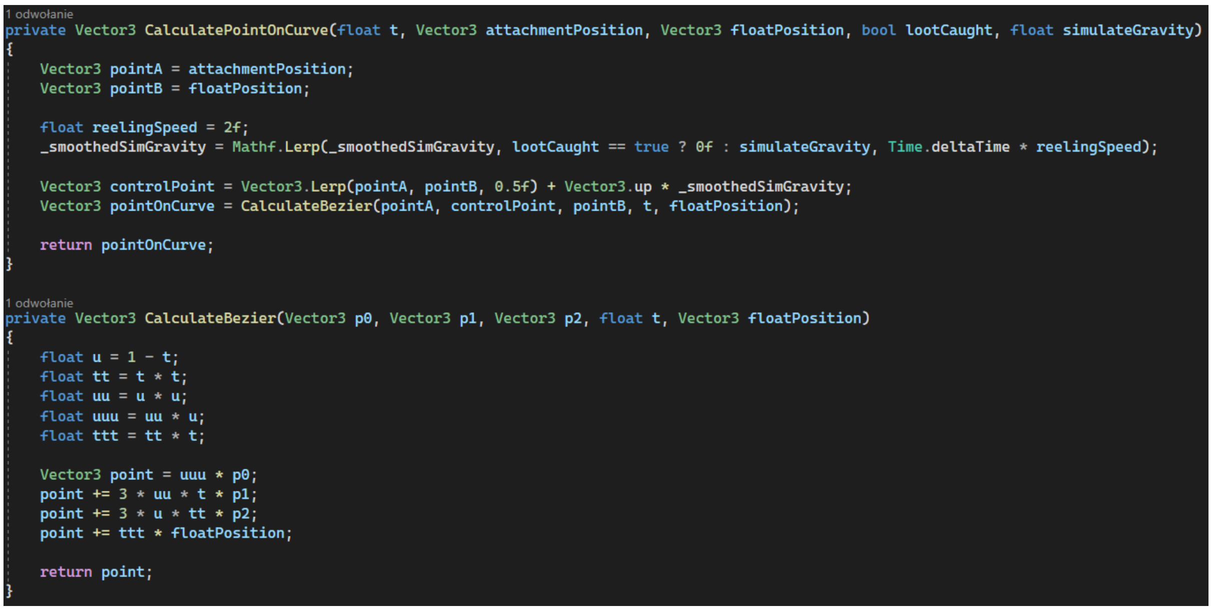
The fishing line is established using the Line Renderer. Initially, we compute the quantity of points comprising the line. This optimization was performed to streamline the number of points essential for displaying the line.



```
1 odwołanie
private static int CalculateLineResolution(float distance, Vector2 resolutionRange)
{
    float minDis = 1f;
    float maxDis = 20f;

    float x = Mathf.InverseLerp(minDis, maxDis, distance);
    float value = Mathf.Lerp(resolutionRange.x, resolutionRange.y, x);

    return (int)value;
}
```

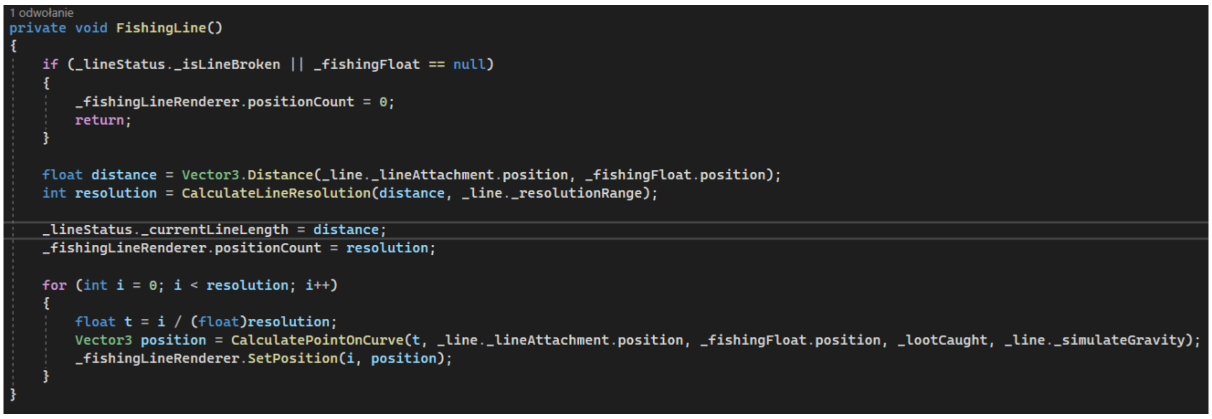
minDis to the minimum distance of the float from the rod. For this value, the number of points is highest to prevent visible transition between points.
maxDis is the inverse of **minDis.**

Subsequently, we determine the positions of these points spanning from the line's attachment point to the float. To imbue a sense of realism into the line, these positions are derived using a Bezier curve calculation.



```
1 odwołanie
private Vector3 CalculatePointOnCurve(float t, Vector3 attachmentPosition, Vector3 floatPosition, bool lootCaught, float simulateGravity)
{
    Vector3 pointA = attachmentPosition;
    Vector3 pointB = floatPosition;

    float reelingSpeed = 2f;
    _smoothedSimGravity = Mathf.Lerp(_smoothedSimGravity, lootCaught == true ? 0f : simulateGravity, Time.deltaTime * reelingSpeed);

    Vector3 controlPoint = Vector3.Lerp(pointA, pointB, 0.5f) + Vector3.up * _smoothedSimGravity;
    Vector3 pointOnCurve = CalculateBezier(pointA, controlPoint, pointB, t, floatPosition);

    return pointOnCurve;
}

1 odwołanie
private Vector3 CalculateBezier(Vector3 p0, Vector3 p1, Vector3 p2, float t, Vector3 floatPosition)
{
    float u = 1 - t;
    float tt = t * t;
    float uu = u * u;
    float uuu = uu * u;
    float ttt = tt * t;

    Vector3 point = uuu * p0;
    point += 3 * uu * t * p1;
    point += 3 * u * tt * p2;
    point += ttt * floatPosition;

    return point;
}
```

The final step is to add line points to the Line Renderer.

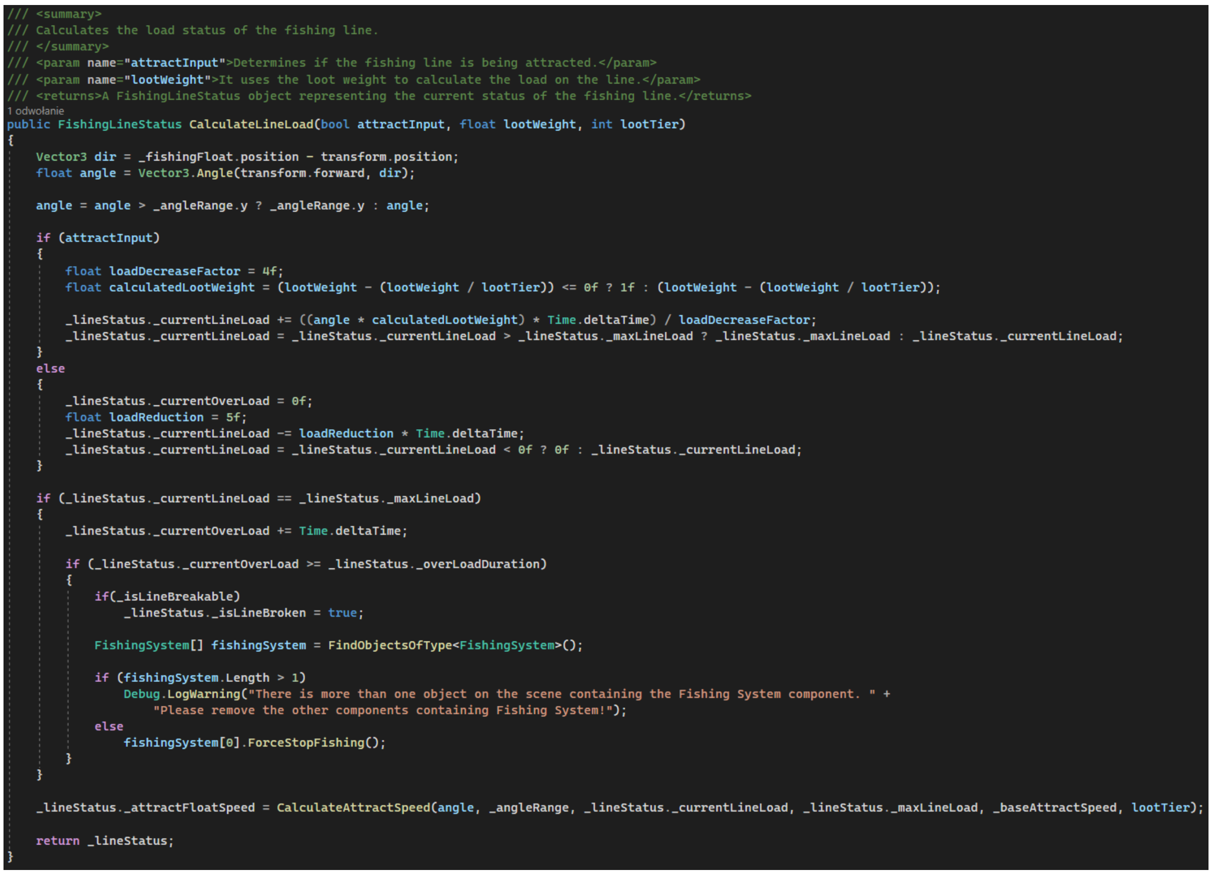


```
1 odwołanie
private void FishingLine()
{
    if (_lineStatus._isLineBroken || _fishingFloat == null)
    {
        _fishingLineRenderer.positionCount = 0;
        return;
    }

    float distance = Vector3.Distance(_line._lineAttachment.position, _fishingFloat.position);
    int resolution = CalculateLineResolution(distance, _line._resolutionRange);

    _lineStatus._currentLineLength = distance;
    _fishingLineRenderer.positionCount = resolution;

    for (int i = 0; i < resolution; i++)
    {
        float t = i / (float)resolution;
        Vector3 position = CalculatePointOnCurve(t, _line._lineAttachment.position, _fishingFloat.position, _lootCaught, _line._simulateGravity);
        _fishingLineRenderer.SetPosition(i, position);
    }
}
```

FISHING GAME TOOL

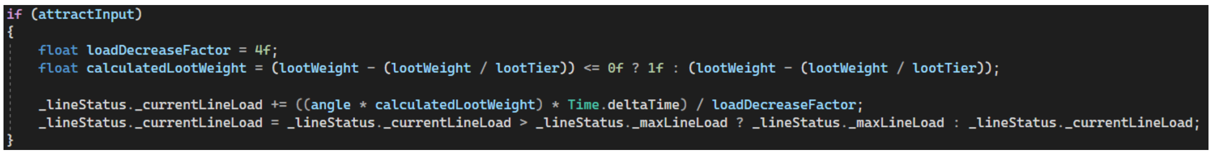
Calculate Line Load

The computation of the line tension is executed utilizing data regarding the catch (including its weight and tier).



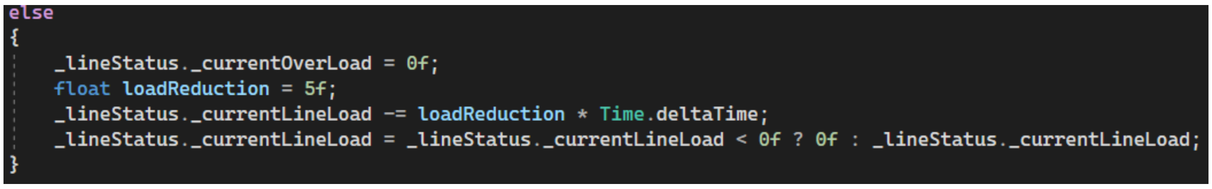
```
/// <summary>
/// Calculates the load status of the fishing line.
/// </summary>
/// <param name="attractInput">Determines if the fishing line is being attracted.</param>
/// <param name="lootWeight">It uses the loot weight to calculate the load on the line.</param>
/// <returns>A FishingLineStatus object representing the current status of the fishing line.</returns>
1 odwołanie
public FishingLineStatus CalculateLineLoad(bool attractInput, float lootWeight, int lootTier)
{
    Vector3 dir = _fishingFloat.position - transform.position;
    float angle = Vector3.Angle(transform.forward, dir);

    angle = angle > _angleRange.y ? _angleRange.y : angle;

    if (attractInput)
    {
        float loadDecreaseFactor = 4f;
        float calculatedLootWeight = (lootWeight - (lootWeight / lootTier)) <= 0f ? 1f : (lootWeight - (lootWeight / lootTier));

        _lineStatus._currentLineLoad += ((angle * calculatedLootWeight) * Time.deltaTime) / loadDecreaseFactor;
        _lineStatus._currentLineLoad = _lineStatus._currentLineLoad > _lineStatus._maxLineLoad ? _lineStatus._maxLineLoad : _lineStatus._currentLineLoad;
    }
    else
    {
        _lineStatus._currentOverLoad = 0f;
        float loadReduction = 5f;
        _lineStatus._currentLineLoad -= loadReduction * Time.deltaTime;
        _lineStatus._currentLineLoad = _lineStatus._currentLineLoad < 0f ? 0f : _lineStatus._currentLineLoad;
    }

    if (_lineStatus._currentLineLoad == _lineStatus._maxLineLoad)
    {
        _lineStatus._currentOverLoad += Time.deltaTime;

        if (_lineStatus._currentOverLoad >= _lineStatus._overLoadDuration)
        {
            if(_isLineBreakable)
                _lineStatus._isLineBroken = true;

            FishingSystem[] fishingSystem = FindObjectsOfType<FishingSystem>();

            if (fishingSystem.Length > 1)
                Debug.LogWarning("There is more than one object on the scene containing the Fishing System component. " +
                    "Please remove the other components containing Fishing System!");
            else
                fishingSystem[0].ForceStopFishing();
        }
    }

    _lineStatus._attractFloatSpeed = CalculateAttractSpeed(angle, _angleRange, _lineStatus._currentLineLoad, _lineStatus._maxLineLoad, _baseAttractSpeed, lootTier);

    return _lineStatus;
}
```

loadDecreaseFactor is used to reduce the rate of load buildup on the line. A lower value speeds up load growth, while a higher value slows it down.



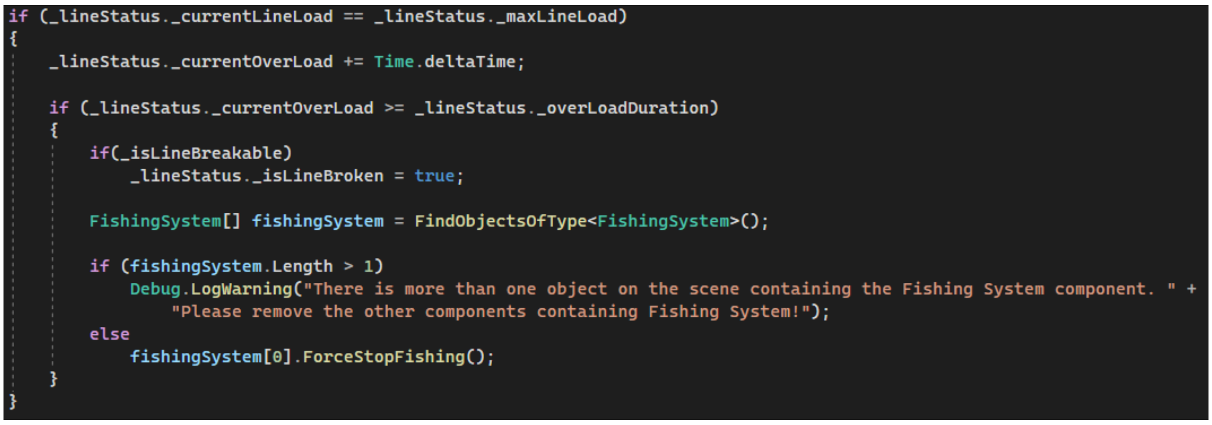
```
if (attractInput)
{
    float loadDecreaseFactor = 4f;
    float calculatedLootWeight = (lootWeight - (lootWeight / lootTier)) <= 0f ? 1f : (lootWeight - (lootWeight / lootTier));

    _lineStatus._currentLineLoad += ((angle * calculatedLootWeight) * Time.deltaTime) / loadDecreaseFactor;
    _lineStatus._currentLineLoad = _lineStatus._currentLineLoad > _lineStatus._maxLineLoad ? _lineStatus._maxLineLoad : _lineStatus._currentLineLoad;
}
```

loadReduction determines how fast the load on the line decreases when the loot pull is let go.



```
else
{
    _lineStatus._currentOverLoad = 0f;
    float loadReduction = 5f;
    _lineStatus._currentLineLoad -= loadReduction * Time.deltaTime;
    _lineStatus._currentLineLoad = _lineStatus._currentLineLoad < 0f ? 0f : _lineStatus._currentLineLoad;
}
```

When the maximum load threshold is surpassed, the countdown for reloading time begins. If this countdown surpasses the predefined threshold, the line will break (assuming this feature is enabled), resulting in the termination of the fishing process.

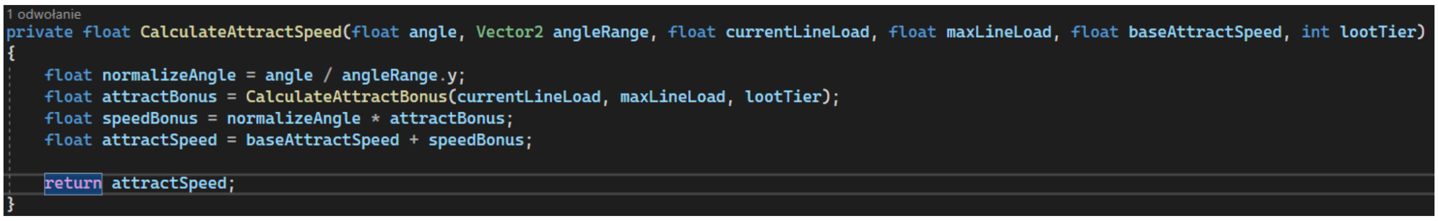


```
if (_lineStatus._currentLineLoad == _lineStatus._maxLineLoad)
{
    _lineStatus._currentOverLoad += Time.deltaTime;

    if (_lineStatus._currentOverLoad >= _lineStatus._overLoadDuration)
    {
        if(_isLineBreakable)
            _lineStatus._isLineBroken = true;

        FishingSystem[] fishingSystem = FindObjectsOfType<FishingSystem>();

        if (fishingSystem.Length > 1)
            Debug.LogWarning("There is more than one object on the scene containing the Fishing System component. " +
                "Please remove the other components containing Fishing System!");
        else
            fishingSystem[0].ForceStopFishing();
    }
}
```

FISHING GAME TOOL

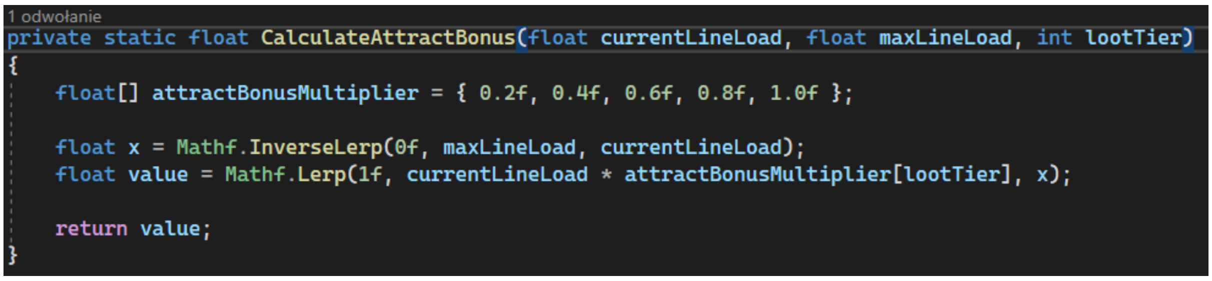
Calculate Attract Speed

The calculation of the attraction speed relies on several interrelated factors. Notably, it considers the angle between the rod and the float; a larger angle corresponds to a higher speed. Furthermore, the current load of the line is also factored in. Sustaining a higher load augments the attraction speed.



```
1 odwołanie
private float CalculateAttractSpeed(float angle, Vector2 angleRange, float currentLineLoad, float maxLineLoad, float baseAttractSpeed, int lootTier)
{
    float normalizeAngle = angle / angleRange.y;
    float attractBonus = CalculateAttractBonus(currentLineLoad, maxLineLoad, lootTier);
    float speedBonus = normalizeAngle * attractBonus;
    float attractSpeed = baseAttractSpeed + speedBonus;

    return attractSpeed;
}
```

Moreover, the loot tier is also leveraged in the calculation. This incorporation allows for the determination of an attraction bonus, with a specific bonus assigned to each loot tier.

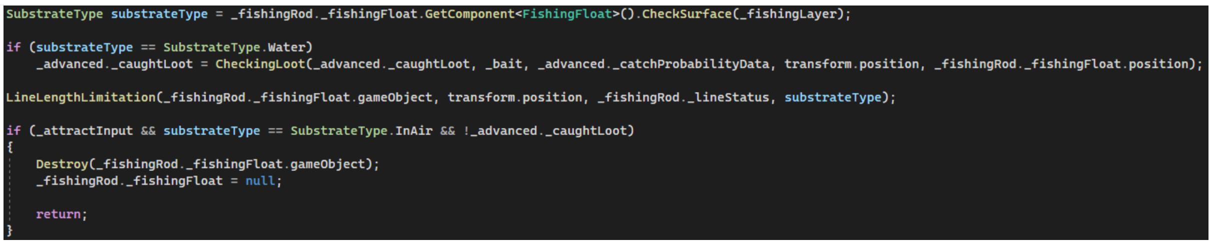


```
1 odwołanie
private static float CalculateAttractBonus(float currentLineLoad, float maxLineLoad, int lootTier)
{
    float[] attractBonusMultiplier = { 0.2f, 0.4f, 0.6f, 0.8f, 1.0f };

    float x = Mathf.InverseLerp(0f, maxLineLoad, currentLineLoad);
    float value = Mathf.Lerp(1f, currentLineLoad * attractBonusMultiplier[lootTier], x);

    return value;
}
```

attractBonusMultiplier is a variable that stores the bonus multiplier. The better the tier, the higher the multiplier.

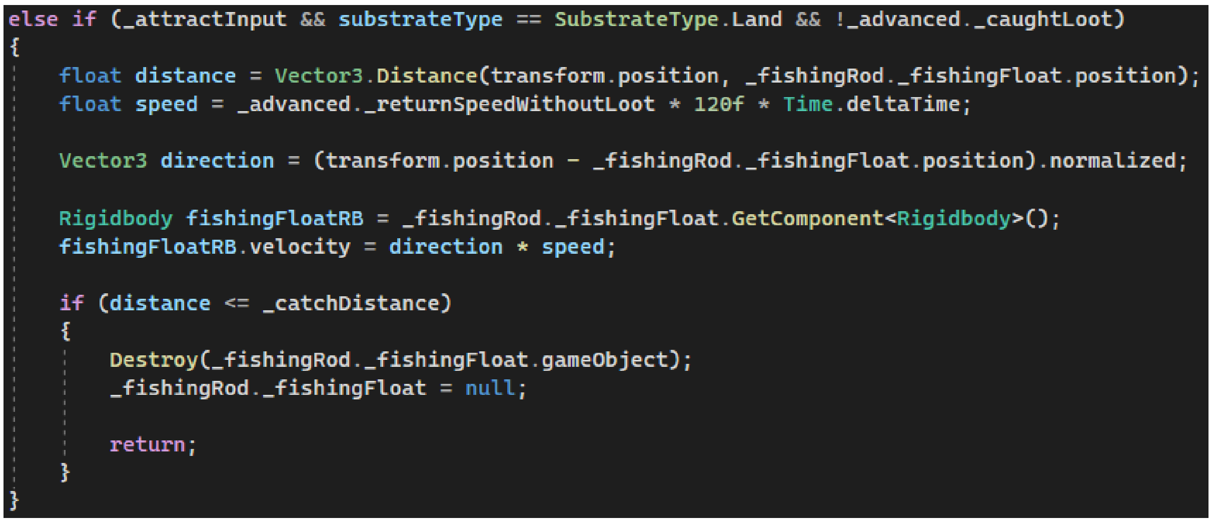
Attract Float

Basic fishing mechanics begin with the Attract Float method. Let's delve into its initialization. Initially, we check the location of the bobber. It can be either in water, on land, or in a state where it hasn't landed anywhere, meaning it's in the air. If it's in the air, it gets attracted instantly.



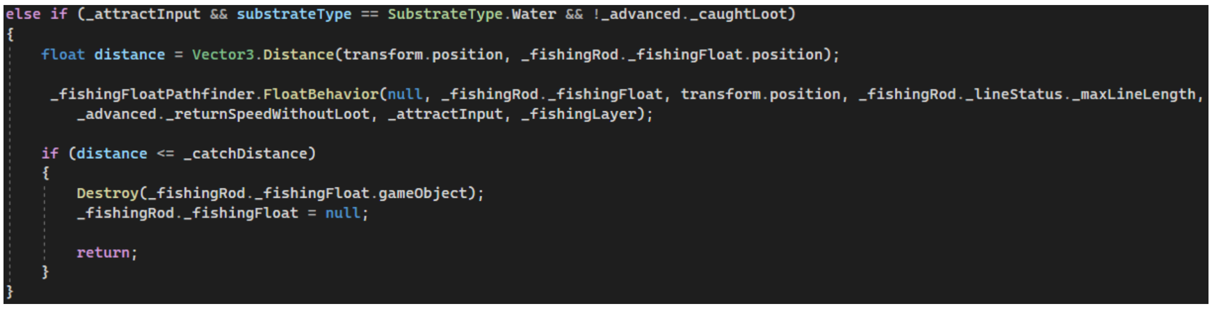
```
SubstrateType substrateType = _fishingRod._fishingFloat.GetComponent<FishingFloat>().CheckSurface(_fishingLayer);

if (substrateType == SubstrateType.Water)
    _advanced._caughtLoot = CheckingLoot(_advanced._caughtLoot, _bait, _advanced._catchProbabilityData, transform.position, _fishingRod._fishingFloat.position);

LineLengthLimitation(_fishingRod._fishingFloat.gameObject, transform.position, _fishingRod._lineStatus, substrateType);

if (_attractInput && substrateType == SubstrateType.InAir && !_advanced._caughtLoot)
{
    Destroy(_fishingRod._fishingFloat.gameObject);
    _fishingRod._fishingFloat = null;

    return;
}
```

The second scenario occurs when the float lands on land. In this case, you can attract it in. In this state, physics affect the float, and it can move objects containing Rigidbody components.



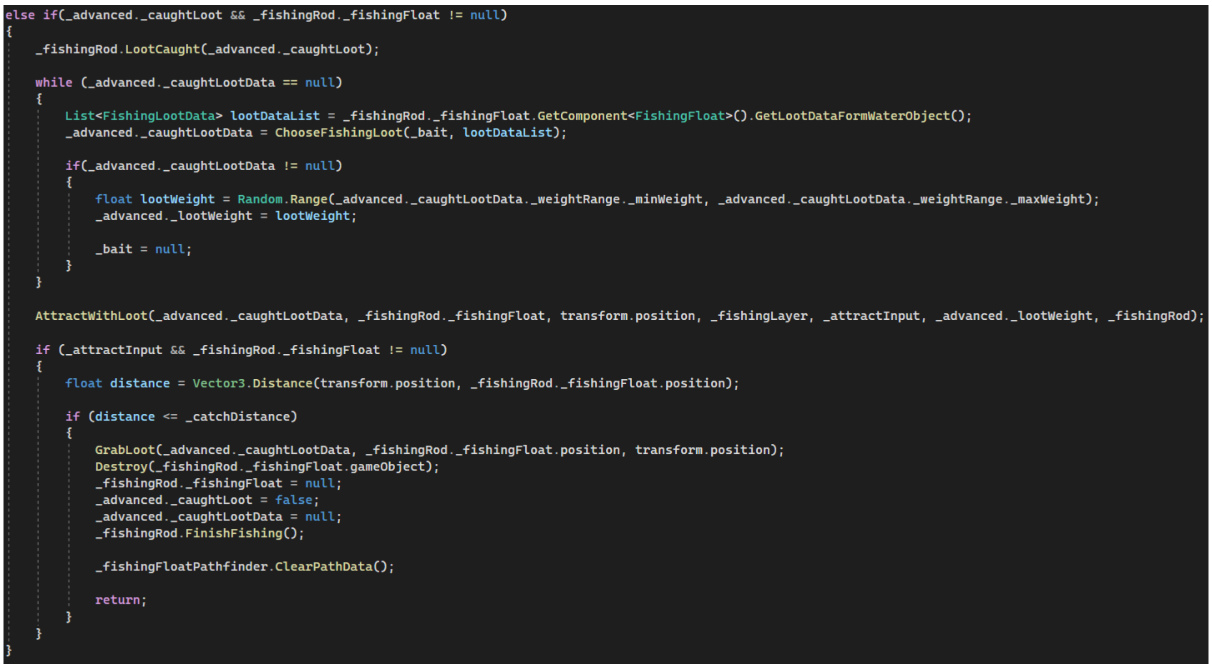
```
else if (_attractInput && substrateType == SubstrateType.Land && !_advanced._caughtLoot)
{
    float distance = Vector3.Distance(transform.position, _fishingRod._fishingFloat.position);
    float speed = _advanced._returnSpeedWithoutLoot * 120f * Time.deltaTime;

    Vector3 direction = (transform.position - _fishingRod._fishingFloat.position).normalized;

    Rigidbody fishingFloatRB = _fishingRod._fishingFloat.GetComponent<Rigidbody>();
    fishingFloatRB.velocity = direction * speed;

    if (distance <= _catchDistance)
    {
        Destroy(_fishingRod._fishingFloat.gameObject);
        _fishingRod._fishingFloat = null;

        return;
    }
}
```

Another scenario occurs when the swimmer successfully lands in the water and the loot remains unclaimed. In such cases, the float becomes attracted. The speed at which a lootless vehicle moves can be configured in **Advanced Settings**.



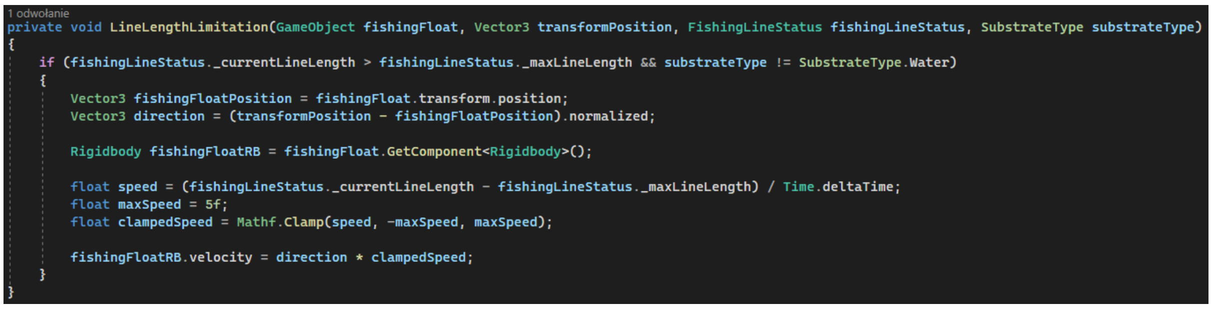
```
else if (_attractInput && substrateType == SubstrateType.Water && !_advanced._caughtLoot)
{
    float distance = Vector3.Distance(transform.position, _fishingRod._fishingFloat.position);

     _fishingFloatPathfinder.FloatBehavior(null, _fishingRod._fishingFloat, transform.position, _fishingRod._lineStatus._maxLineLength,
        _advanced._returnSpeedWithoutLoot, _attractInput, _fishingLayer);

    if (distance <= _catchDistance)
    {
        Destroy(_fishingRod._fishingFloat.gameObject);
        _fishingRod._fishingFloat = null;

        return;
    }
}
```

The third scenario occurs when the float is situated in the water and the loot has been successfully caught.



```
else if(_advanced._caughtLoot && _fishingRod._fishingFloat != null)
{
    _fishingRod.LootCaught(_advanced._caughtLoot);

    while (_advanced._caughtLootData == null)
    {
        List<FishingLootData> lootDataList = _fishingRod._fishingFloat.GetComponent<FishingFloat>().GetLootDataFormWaterObject();
        _advanced._caughtLootData = ChooseFishingLoot(_bait, lootDataList);

        if(_advanced._caughtLootData != null)
        {
            float lootWeight = Random.Range(_advanced._caughtLootData._weightRange._minWeight, _advanced._caughtLootData._weightRange._maxWeight);
            _advanced._lootWeight = lootWeight;

            _bait = null;
        }
    }

    AttractWithLoot(_advanced._caughtLootData, _fishingRod._fishingFloat, transform.position, _fishingLayer, _attractInput, _advanced._lootWeight, _fishingRod);

    if (_attractInput && _fishingRod._fishingFloat != null)
    {
        float distance = Vector3.Distance(transform.position, _fishingRod._fishingFloat.position);

        if (distance <= _catchDistance)
        {
            GrabLoot(_advanced._caughtLootData, _fishingRod._fishingFloat.position, transform.position);
            Destroy(_fishingRod._fishingFloat.gameObject);
            _fishingRod._fishingFloat = null;
            _advanced._caughtLoot = false;
            _advanced._caughtLootData = null;
            _fishingRod.FinishFishing();

            _fishingFloatPathfinder.ClearPathData();

            return;
        }
    }
}
```

The **Line Length Limitation** method aims to limit the casting distance of the float when the length of the fishing line has been exceeded. It also moves the float when it has been cast on land and the player begins to move away from it.

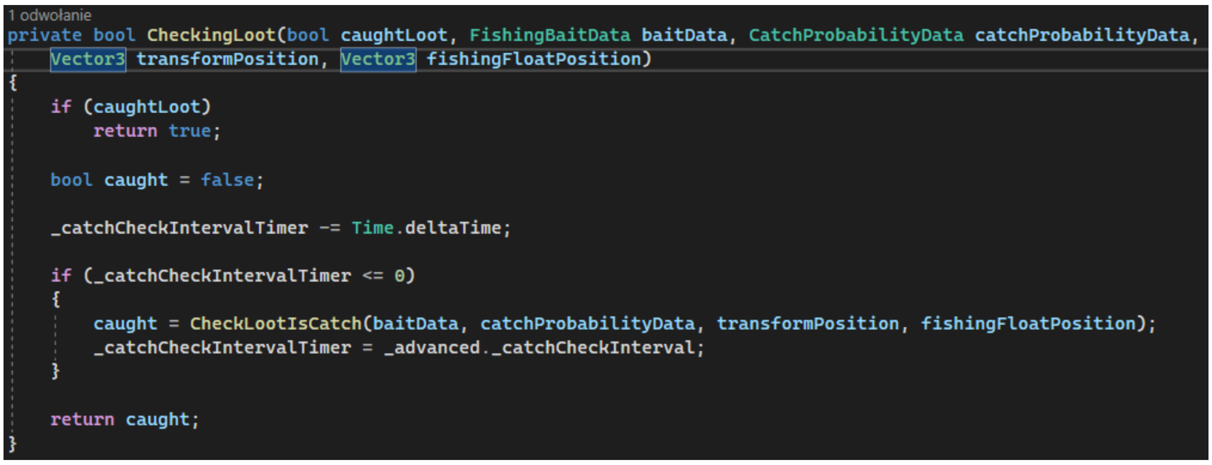


```
1 odwołanie
private void LineLengthLimitation(GameObject fishingFloat, Vector3 transformPosition, FishingLineStatus fishingLineStatus, SubstrateType substrateType)
{
    if (fishingLineStatus._currentLineLength > fishingLineStatus._maxLineLength && substrateType != SubstrateType.Water)
    {
        Vector3 fishingFloatPosition = fishingFloat.transform.position;
        Vector3 direction = (transformPosition - fishingFloatPosition).normalized;

        Rigidbody fishingFloatRB = fishingFloat.GetComponent<Rigidbody>();

        float speed = (fishingLineStatus._currentLineLength - fishingLineStatus._maxLineLength) / Time.deltaTime;
        float maxSpeed = 5f;
        float clampedSpeed = Mathf.Clamp(speed, -maxSpeed, maxSpeed);

        fishingFloatRB.velocity = direction * clampedSpeed;
    }
}
```

FISHING GAME TOOL

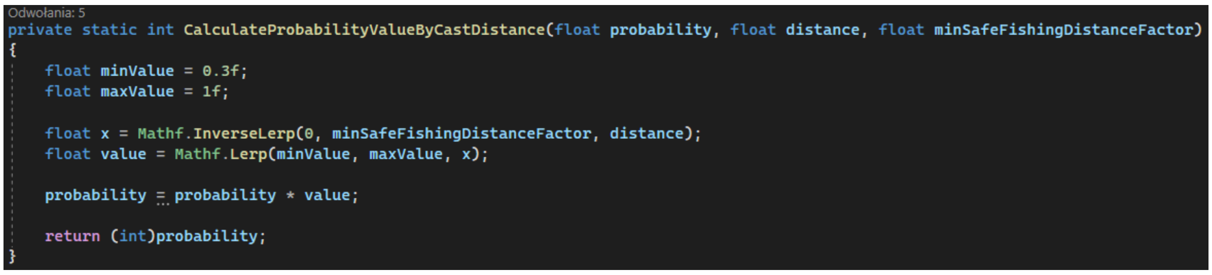
Checking Loot

The verification of whether the loot has been caught takes place at predefined intervals. Should the loot have already been secured, the method returns a "true" value.

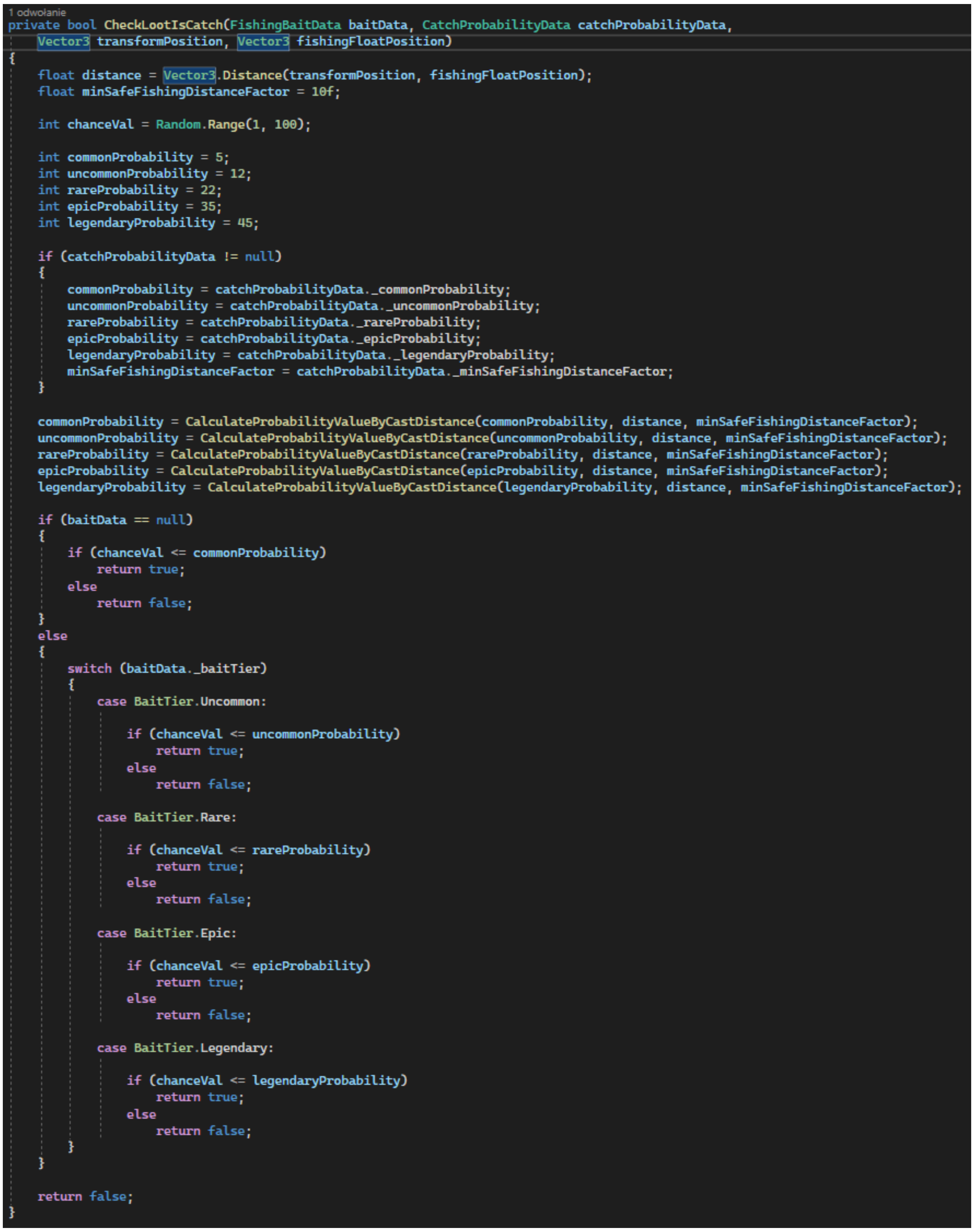


```
1 odwołanie
private bool CheckingLoot(bool caughtLoot, FishingBaitData baitData, CatchProbabilityData catchProbabilityData,
    Vector3 transformPosition, Vector3 fishingFloatPosition)
{
    if (caughtLoot)
        return true;

    bool caught = false;

    _catchCheckIntervalTimer -= Time.deltaTime;

    if (_catchCheckIntervalTimer <= 0)
    {
        caught = CheckLootIsCatch(baitData, catchProbabilityData, transformPosition, fishingFloatPosition);
        _catchCheckIntervalTimer = _advanced._catchCheckInterval;
    }

    return caught;
}
```

The mechanism for determining whether the loot has been captured relies on probability assessment. A random number ranging from 1 to 100 is generated. If this drawn number falls within a specified range, the method returns a "true" value. The range of this probability is expanded by incorporating bait. The tier of the bait influences the extent of this range. This process involves utilizing a customized Catch Probability Data structure.
The probability calculation for catching also takes into account the distance at which the float has been cast from the fishing rod.



```
Odwołania: 5
private static int CalculateProbabilityValueByCastDistance(float probability, float distance, float minSafeFishingDistanceFactor)
{
    float minValue = 0.3f;
    float maxValue = 1f;

    float x = Mathf.InverseLerp(0, minSafeFishingDistanceFactor, distance);
    float value = Mathf.Lerp(minValue, maxValue, x);

    probability = probability * value;

    return (int)probability;
}
```

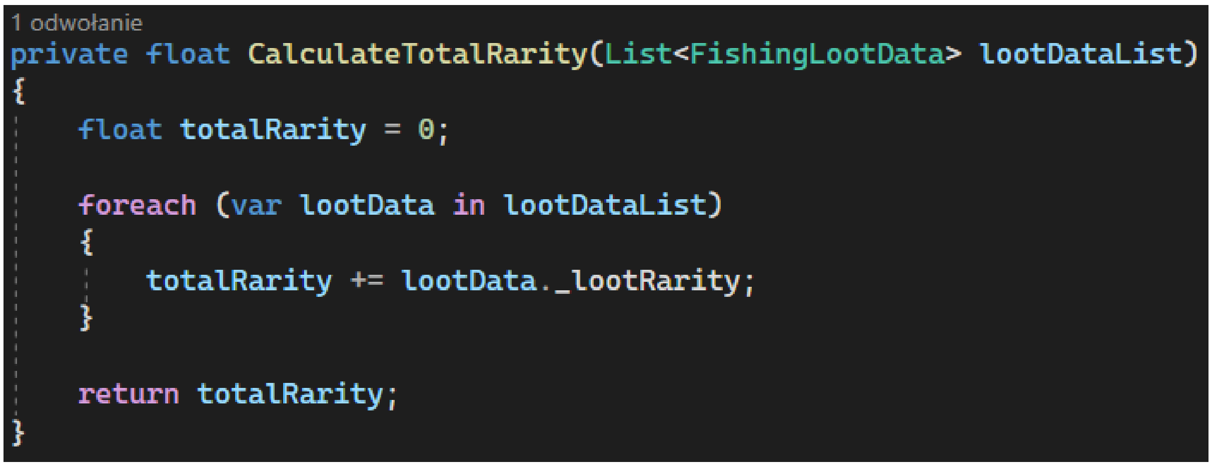


```
1 odwołanie
private bool CheckLootIsCatch(FishingBaitData baitData, CatchProbabilityData catchProbabilityData,
    Vector3 transformPosition, Vector3 fishingFloatPosition)
{
    float distance = Vector3.Distance(transformPosition, fishingFloatPosition);
    float minSafeFishingDistanceFactor = 10f;

    int chanceVal = Random.Range(1, 100);

    int commonProbability = 5;
    int uncommonProbability = 12;
    int rareProbability = 22;
    int epicProbability = 35;
    int legendaryProbability = 45;

    if (catchProbabilityData != null)
    {
        commonProbability = catchProbabilityData._commonProbability;
        uncommonProbability = catchProbabilityData._uncommonProbability;
        rareProbability = catchProbabilityData._rareProbability;
        epicProbability = catchProbabilityData._epicProbability;
        legendaryProbability = catchProbabilityData._legendaryProbability;
        minSafeFishingDistanceFactor = catchProbabilityData._minSafeFishingDistanceFactor;
    }

    commonProbability = CalculateProbabilityValueByCastDistance(commonProbability, distance, minSafeFishingDistanceFactor);
    uncommonProbability = CalculateProbabilityValueByCastDistance(uncommonProbability, distance, minSafeFishingDistanceFactor);
    rareProbability = CalculateProbabilityValueByCastDistance(rareProbability, distance, minSafeFishingDistanceFactor);
    epicProbability = CalculateProbabilityValueByCastDistance(epicProbability, distance, minSafeFishingDistanceFactor);
    legendaryProbability = CalculateProbabilityValueByCastDistance(legendaryProbability, distance, minSafeFishingDistanceFactor);

    if (baitData == null)
    {
        if (chanceVal <= commonProbability)
            return true;
        else
            return false;
    }
    else
    {
        switch (baitData._baitTier)
        {
            case BaitTier.Uncommon:

                if (chanceVal <= uncommonProbability)
                    return true;
                else
                    return false;

            case BaitTier.Rare:

                if (chanceVal <= rareProbability)
                    return true;
                else
                    return false;

            case BaitTier.Epic:

                if (chanceVal <= epicProbability)
                    return true;
                else
                    return false;

            case BaitTier.Legendary:

                if (chanceVal <= legendaryProbability)
                    return true;
                else
                    return false;
        }
    }

    return false;
}
```

FISHING GAME TOOL

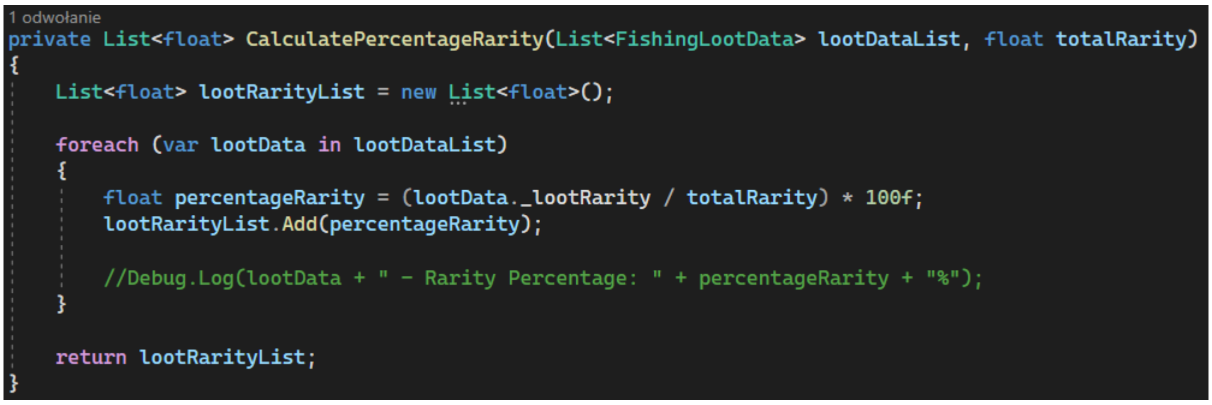
Choose Fishing Loot

The selection of loot is conducted from a pre-existing database containing available items within a specific fishing location. The probability mechanism and the rarity of the items collectively influence the determination of the captured loot. Initially, the rarity of all items is recalibrated and standardized to a scale ranging from 0 to 100. This process is automated, eliminating the need for manual rarity calculations for individual items to ensure their collective sum is 100. The procedure commences with the computation of the cumulative rarity value encompassing all available loot items.



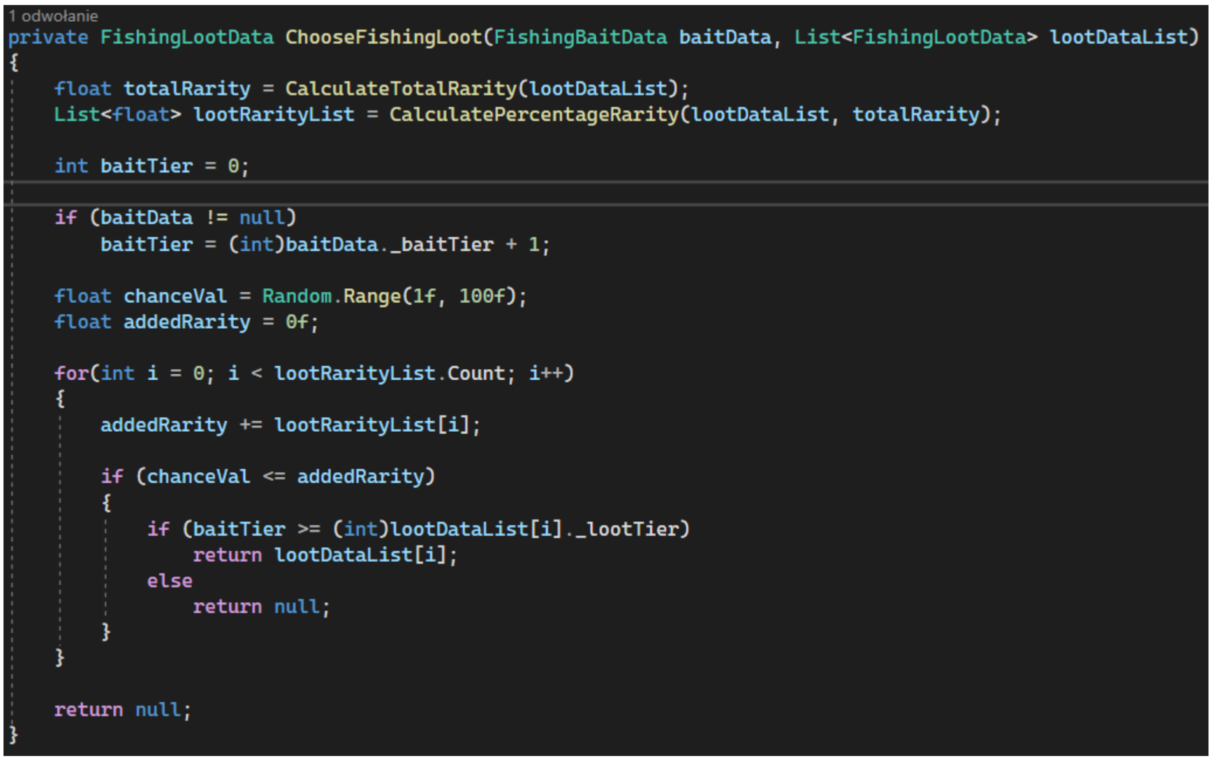
```
1 odwołanie
private float CalculateTotalRarity(List<FishingLootData> lootDataList)
{
    float totalRarity = 0;

    foreach (var lootData in lootDataList)
    {
        totalRarity += lootData._lootRarity;
    }

    return totalRarity;
}
```

Then we recalculate everything so that their total is 100.



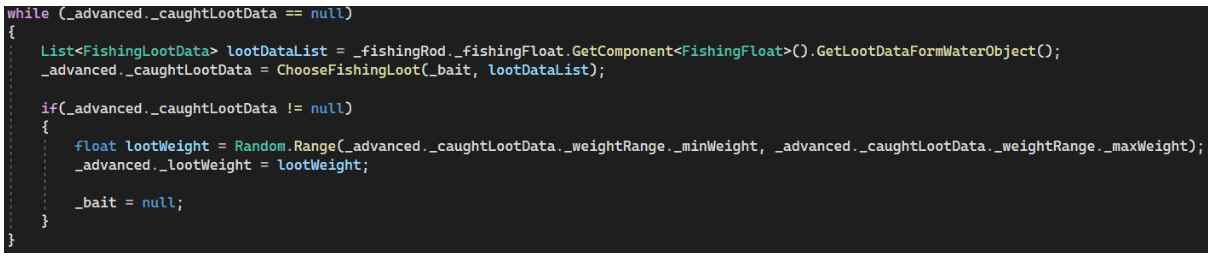
```
1 odwołanie
private List<float> CalculatePercentageRarity(List<FishingLootData> lootDataList, float totalRarity)
{
    List<float> lootRarityList = new List<float>();

    foreach (var lootData in lootDataList)
    {
        float percentageRarity = (lootData._lootRarity / totalRarity) * 100f;
        lootRarityList.Add(percentageRarity);

        //Debug.Log(lootData + " - Rarity Percentage: " + percentageRarity + "%");
    }

    return lootRarityList;
}
```

The concluding stage involves selecting the appropriate loot. A random number is generated, and if it falls within the predefined range, the loot's tier is evaluated. It's essential that the loot's tier does not exceed the tier of the lure. This mechanism is designed to add a distinctive aspect to loot with higher tiers – they can exclusively be captured using the suitable bait, thereby enhancing their uniqueness.



```
1 odwołanie
private FishingLootData ChooseFishingLoot(FishingBaitData baitData, List<FishingLootData> lootDataList)
{
    float totalRarity = CalculateTotalRarity(lootDataList);
    List<float> lootRarityList = CalculatePercentageRarity(lootDataList, totalRarity);

    int baitTier = 0;

    if (baitData != null)
        baitTier = (int)baitData._baitTier + 1;

    float chanceVal = Random.Range(1f, 100f);
    float addedRarity = 0f;

    for(int i = 0; i < lootRarityList.Count; i++)
    {
        addedRarity += lootRarityList[i];

        if (chanceVal <= addedRarity)
        {
            if (baitTier >= (int)lootDataList[i]._lootTier)
                return lootDataList[i];
            else
                return null;
        }
    }

    return null;
}
```

In case no loot is chosen, the entire loot selection process will be reattempted in the subsequent iteration of the while loop.

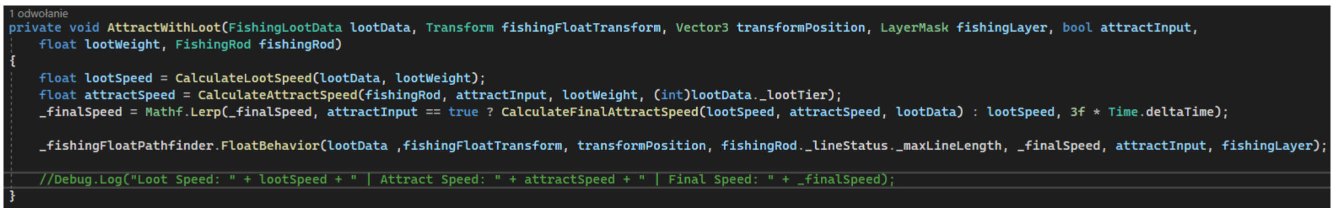


```
while (_advanced._caughtLootData == null)
{
    List<FishingLootData> lootDataList = _fishingRod._fishingFloat.GetComponent<FishingFloat>().GetLootDataFormWaterObject();
    _advanced._caughtLootData = ChooseFishingLoot(_bait, lootDataList);

    if(_advanced._caughtLootData != null)
    {
        float lootWeight = Random.Range(_advanced._caughtLootData._weightRange._minWeight, _advanced._caughtLootData._weightRange._maxWeight);
        _advanced._lootWeight = lootWeight;

        _bait = null;
    }
}
```

FISHING GAME TOOL

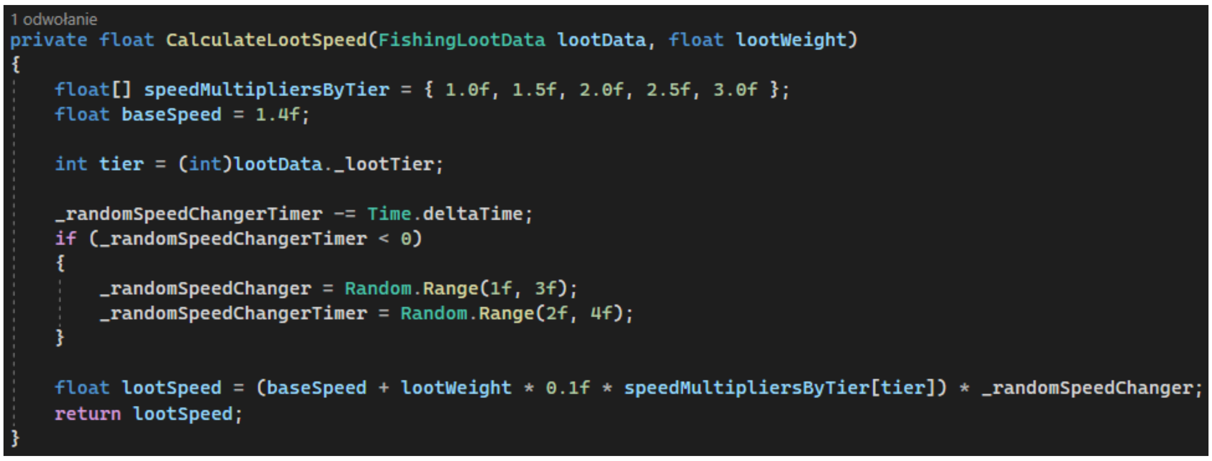
Attract With Loot

This method is responsible for calculating loot speed, attraction and final speed. The data is transferred to **Fishing Float Pathfinder**, which is responsible for the behavior and attraction of the float.



```
1 odwołanie
private void AttractWithLoot(FishingLootData lootData, Transform fishingFloatTransform, Vector3 transformPosition, LayerMask fishingLayer, bool attractInput,
    float lootWeight, FishingRod fishingRod)
{
    float lootSpeed = CalculateLootSpeed(lootData, lootWeight);
    float attractSpeed = CalculateAttractSpeed(fishingRod, attractInput, lootWeight, (int)lootData._lootTier);
    _finalSpeed = Mathf.Lerp(_finalSpeed, attractInput == true ? CalculateFinalAttractSpeed(lootSpeed, attractSpeed, lootData) : lootSpeed, 3f * Time.deltaTime);

    _fishingFloatPathfinder.FloatBehavior(lootData ,fishingFloatTransform, transformPosition, fishingRod._lineStatus._maxLineLength, _finalSpeed, attractInput, fishingLayer);

    //Debug.Log("Loot Speed: " + lootSpeed + " | Attract Speed: " + attractSpeed + " | Final Speed: " + _finalSpeed);
}
```

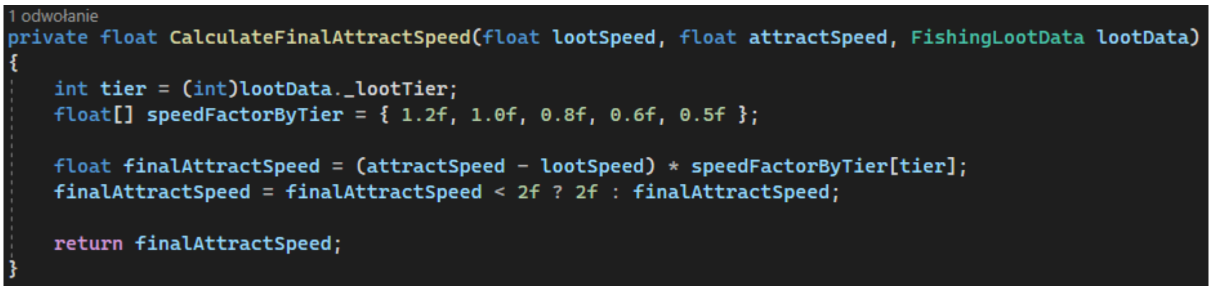
CalculateLootSpeed method is responsible for calculating the speed of the loot (fish).



```
1 odwołanie
private float CalculateLootSpeed(FishingLootData lootData, float lootWeight)
{
    float[] speedMultipliersByTier = { 1.0f, 1.5f, 2.0f, 2.5f, 3.0f };
    float baseSpeed = 1.4f;

    int tier = (int)lootData._lootTier;

    _randomSpeedChangerTimer -= Time.deltaTime;
    if (_randomSpeedChangerTimer < 0)
    {
        _randomSpeedChanger = Random.Range(1f, 3f);
        _randomSpeedChangerTimer = Random.Range(2f, 4f);
    }

    float lootSpeed = (baseSpeed + lootWeight * 0.1f * speedMultipliersByTier[tier]) * _randomSpeedChanger;
    return lootSpeed;
}
```

speedMultipliersByTier is a array of speed multipliers based on loot tier.
baseSpeed is the base speed of the loot.

CalculateFinalAttractSpeed method is responsible for calculating the final loot speed and attraction. It brings the loot speed and the attraction speed to a single speed.



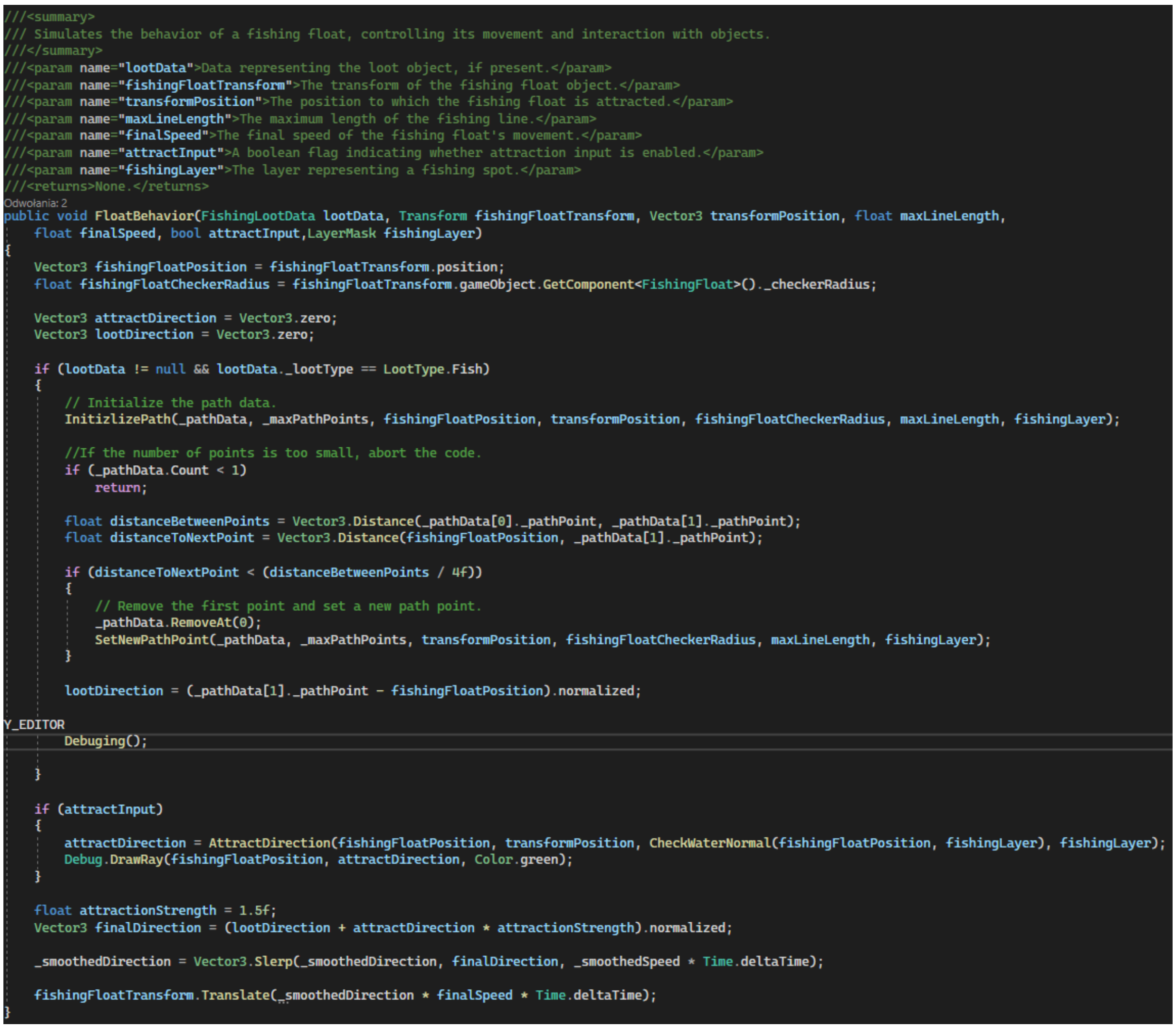
```
1 odwołanie
private float CalculateFinalAttractSpeed(float lootSpeed, float attractSpeed, FishingLootData lootData)
{
    int tier = (int)lootData._lootTier;
    float[] speedFactorByTier = { 1.2f, 1.0f, 0.8f, 0.6f, 0.5f };

    float finalAttractSpeed = (attractSpeed - lootSpeed) * speedFactorByTier[tier];
    finalAttractSpeed = finalAttractSpeed < 2f ? 2f : finalAttractSpeed;

    return finalAttractSpeed;
}
```

speedFactorByTier is a list of speed multipliers that determine how much slower the attract speed will be based on the loot tier.

FISHING GAME TOOL

Float Behavior

FloatBehavior (formerly known as **GetPointForFloat**) is a method associated with the **FishingFloatPathfinder** class (formerly known as **FishingFloatBehavior**). The purpose of this class is to impart movement to the fishing float. It simulates fish behavior when the loot type is appropriate and is responsible for attracting the fishing float when it is in the water. This system completely changes the simulation of fish behavior. It no longer returns any values. The system that generates the destination point has been replaced by a system that creates a path along which the fishing float moves. This approach results in fewer issues.

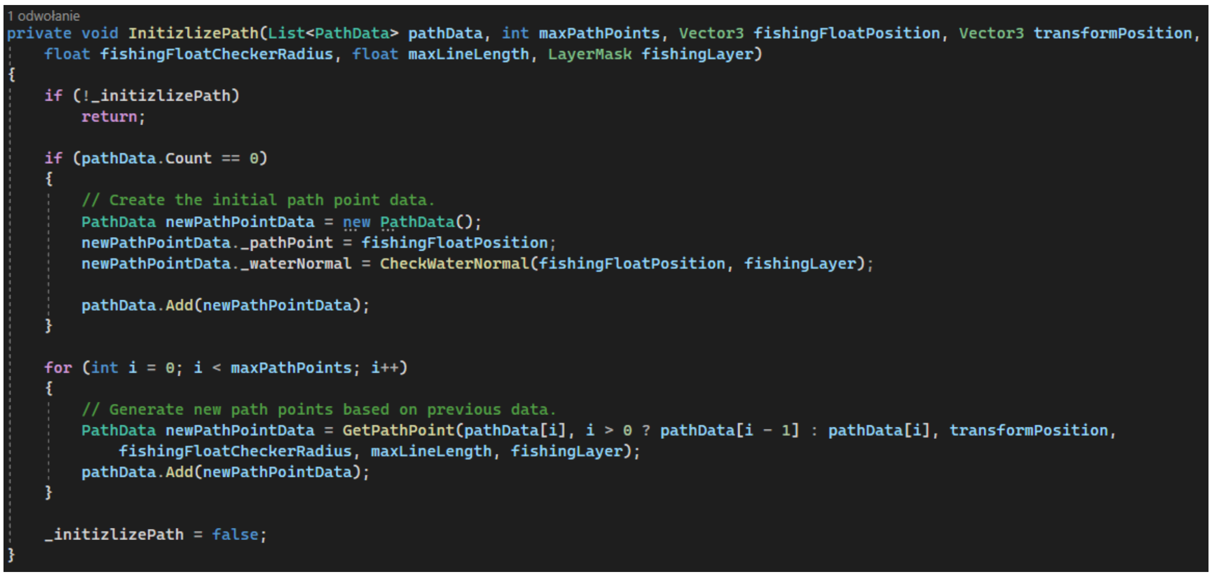


```
///<summary>
/// Simulates the behavior of a fishing float, controlling its movement and interaction with objects.
///</summary>
///<param name="lootData">Data representing the loot object, if present.</param>
///<param name="fishingFloatTransform">The transform of the fishing float object.</param>
///<param name="transformPosition">The position to which the fishing float is attracted.</param>
///<param name="maxLineLength">The maximum length of the fishing line.</param>
///<param name="finalSpeed">The final speed of the fishing float's movement.</param>
///<param name="attractInput">A boolean flag indicating whether attraction input is enabled.</param>
///<param name="fishingLayer">The layer representing a fishing spot.</param>
///<returns>None.</returns>
Odwołania: 2
public void FloatBehavior(FishingLootData lootData, Transform fishingFloatTransform, Vector3 transformPosition, float maxLineLength,
    float finalSpeed, bool attractInput,LayerMask fishingLayer)
{
    Vector3 fishingFloatPosition = fishingFloatTransform.position;
    float fishingFloatCheckerRadius = fishingFloatTransform.gameObject.GetComponent<FishingFloat>()._checkerRadius;

    Vector3 attractDirection = Vector3.zero;
    Vector3 lootDirection = Vector3.zero;

    if (lootData != null && lootData._lootType == LootType.Fish)
    {
        // Initialize the path data.
        InitizlizePath(_pathData, _maxPathPoints, fishingFloatPosition, transformPosition, fishingFloatCheckerRadius, maxLineLength, fishingLayer);

        //If the number of points is too small, abort the code.
        if (_pathData.Count < 1)
            return;

        float distanceBetweenPoints = Vector3.Distance(_pathData[0]._pathPoint, _pathData[1]._pathPoint);
        float distanceToNextPoint = Vector3.Distance(fishingFloatPosition, _pathData[1]._pathPoint);

        if (distanceToNextPoint < (distanceBetweenPoints / 4f))
        {
            // Remove the first point and set a new path point.
            _pathData.RemoveAt(0);
            SetNewPathPoint(_pathData, _maxPathPoints, transformPosition, fishingFloatCheckerRadius, maxLineLength, fishingLayer);
        }

        lootDirection = (_pathData[1]._pathPoint - fishingFloatPosition).normalized;

Y_EDITOR
        Debuging();

    }

    if (attractInput)
    {
        attractDirection = AttractDirection(fishingFloatPosition, transformPosition, CheckWaterNormal(fishingFloatPosition, fishingLayer), fishingLayer);
        Debug.DrawRay(fishingFloatPosition, attractDirection, Color.green);
    }

    float attractionStrength = 1.5f;
    Vector3 finalDirection = (lootDirection + attractDirection * attractionStrength).normalized;

    _smoothedDirection = Vector3.Slerp(_smoothedDirection, finalDirection, _smoothedSpeed * Time.deltaTime);

    fishingFloatTransform.Translate(_smoothedDirection * finalSpeed * Time.deltaTime);
}
```

FISHING GAME TOOL

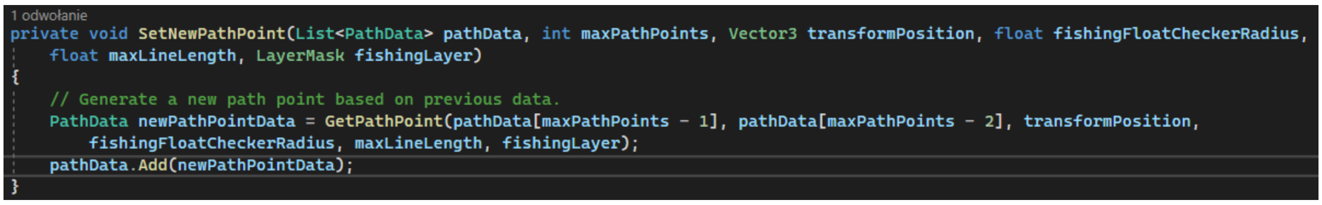
Initialize Path And Set New Path Point

InitializePath method is responsible for initializing the path of the fishing float. It is called as the first step when the loot (fish) is caught on the hook.



```
1 odwołanie
private void InitizlizePath(List<PathData> pathData, int maxPathPoints, Vector3 fishingFloatPosition, Vector3 transformPosition,
    float fishingFloatCheckerRadius, float maxLineLength, LayerMask fishingLayer)
{
    if (!_initizlizePath)
        return;

    if (pathData.Count == 0)
    {
        // Create the initial path point data.
        PathData newPathPointData = new PathData();
        newPathPointData._pathPoint = fishingFloatPosition;
        newPathPointData._waterNormal = CheckWaterNormal(fishingFloatPosition, fishingLayer);

        pathData.Add(newPathPointData);
    }

    for (int i = 0; i < maxPathPoints; i++)
    {
        // Generate new path points based on previous data.
        PathData newPathPointData = GetPathPoint(pathData[i], i > 0 ? pathData[i - 1] : pathData[i], transformPosition,
            fishingFloatCheckerRadius, maxLineLength, fishingLayer);
        pathData.Add(newPathPointData);
    }

    _initizlizePath = false;
}
```

SetNewPathPoint method is responsible for generating a new point the path when the fishing float reaches the next point.

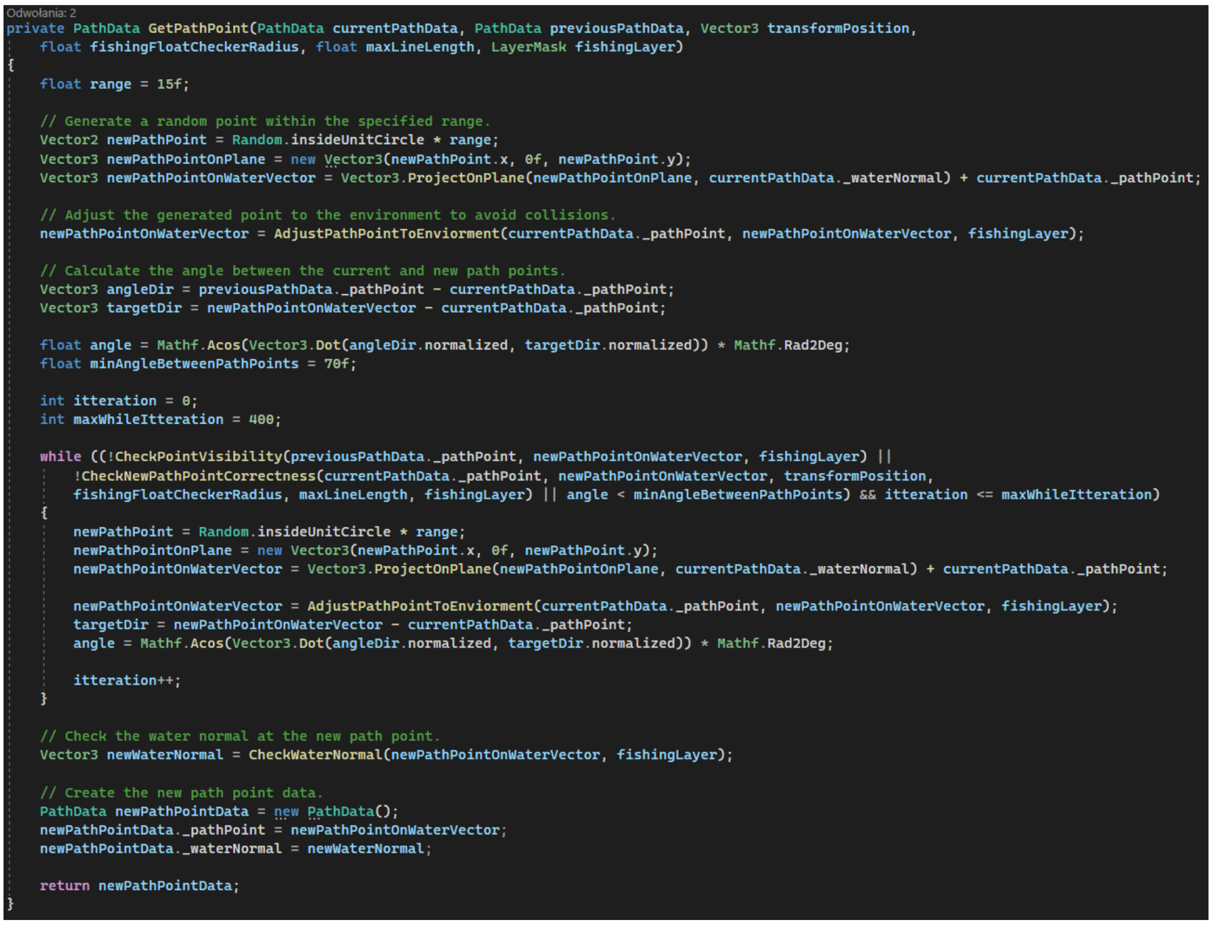


```
1 odwołanie
private void SetNewPathPoint(List<PathData> pathData, int maxPathPoints, Vector3 transformPosition, float fishingFloatCheckerRadius,
    float maxLineLength, LayerMask fishingLayer)
{
    // Generate a new path point based on previous data.
    PathData newPathPointData = GetPathPoint(pathData[maxPathPoints - 1], pathData[maxPathPoints - 2], transformPosition,
        fishingFloatCheckerRadius, maxLineLength, fishingLayer);
    pathData.Add(newPathPointData);
}
```

FISHING GAME TOOL

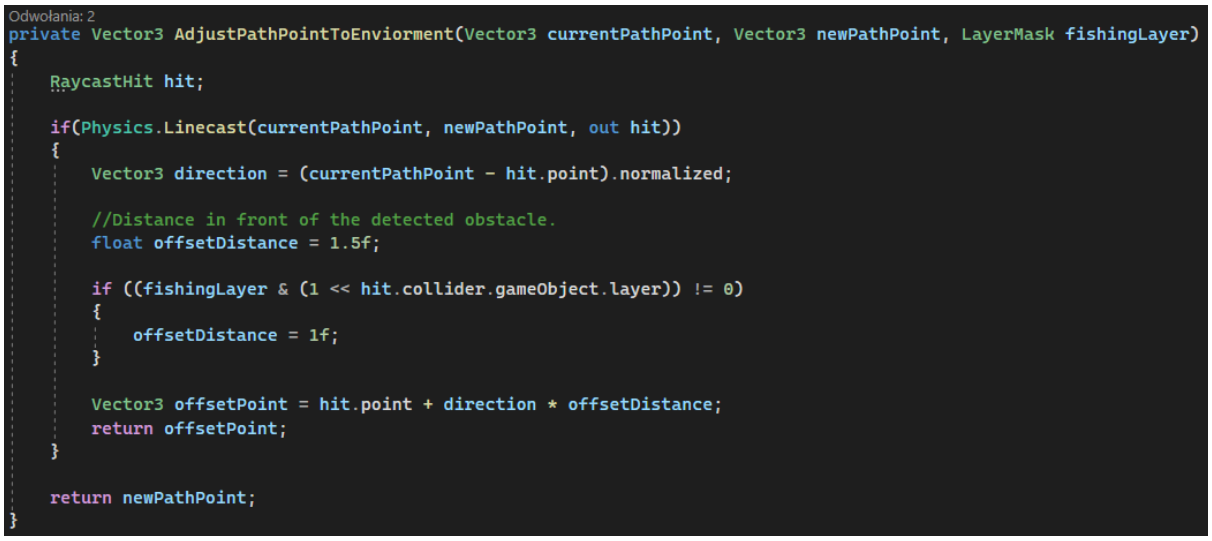
Get Path Point

GetPathPoint method is the primary function responsible for generating path points. It creates a new point on the appropriate water plane, adjusts it to the terrain using Raycast, and checks its validity within a while loop. Full accuracy is not necessary, and to optimize and guard against potential issues with freezing, the loop has a predefined number of iterations it can perform. If it doesn't find a suitable point within this limit, the last checked point is used. Additionally, the water normal beneath the created point is examined, which is used in generating the next point. Finally, a new **PathData** is created, storing the point's position and the water normal beneath it.



```
Odwołania: 2
private PathData GetPathPoint(PathData currentPathData, PathData previousPathData, Vector3 transformPosition,
    float fishingFloatCheckerRadius, float maxLineLength, LayerMask fishingLayer)
{
    float range = 15f;

    // Generate a random point within the specified range.
    Vector2 newPathPoint = Random.insideUnitCircle * range;
    Vector3 newPathPointOnPlane = new Vector3(newPathPoint.x, 0f, newPathPoint.y);
    Vector3 newPathPointOnWaterVector = Vector3.ProjectOnPlane(newPathPointOnPlane, currentPathData._waterNormal) + currentPathData._pathPoint;

    // Adjust the generated point to the environment to avoid collisions.
    newPathPointOnWaterVector = AdjustPathPointToEnviorment(currentPathData._pathPoint, newPathPointOnWaterVector, fishingLayer);

    // Calculate the angle between the current and new path points.
    Vector3 angleDir = previousPathData._pathPoint - currentPathData._pathPoint;
    Vector3 targetDir = newPathPointOnWaterVector - currentPathData._pathPoint;

    float angle = Mathf.Acos(Vector3.Dot(angleDir.normalized, targetDir.normalized)) * Mathf.Rad2Deg;
    float minAngleBetweenPathPoints = 70f;

    int itteration = 0;
    int maxWhileItteration = 400;

    while ((!CheckPointVisibility(previousPathData._pathPoint, newPathPointOnWaterVector, fishingLayer) ||
        !CheckNewPathPointCorrectness(currentPathData._pathPoint, newPathPointOnWaterVector, transformPosition,
        fishingFloatCheckerRadius, maxLineLength, fishingLayer) || angle < minAngleBetweenPathPoints) && itteration <= maxWhileItteration)
    {
        newPathPoint = Random.insideUnitCircle * range;
        newPathPointOnPlane = new Vector3(newPathPoint.x, 0f, newPathPoint.y);
        newPathPointOnWaterVector = Vector3.ProjectOnPlane(newPathPointOnPlane, currentPathData._waterNormal) + currentPathData._pathPoint;

        newPathPointOnWaterVector = AdjustPathPointToEnviorment(currentPathData._pathPoint, newPathPointOnWaterVector, fishingLayer);
        targetDir = newPathPointOnWaterVector - currentPathData._pathPoint;
        angle = Mathf.Acos(Vector3.Dot(angleDir.normalized, targetDir.normalized)) * Mathf.Rad2Deg;

        itteration++;
    }

    // Check the water normal at the new path point.
    Vector3 newWaterNormal = CheckWaterNormal(newPathPointOnWaterVector, fishingLayer);

    // Create the new path point data.
    PathData newPathPointData = new PathData();
    newPathPointData._pathPoint = newPathPointOnWaterVector;
    newPathPointData._waterNormal = newWaterNormal;

    return newPathPointData;
}
```

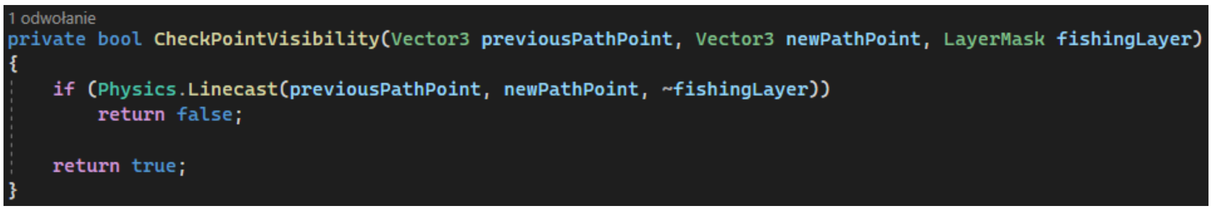
AdjustPathPointToEnvironment method checks whether the generated point might be obscured by any terrain objects. If it is, the method relocates the point closer to ensure visibility.



```
Odwołania: 2
private Vector3 AdjustPathPointToEnviorment(Vector3 currentPathPoint, Vector3 newPathPoint, LayerMask fishingLayer)
{
    RaycastHit hit;

    if(Physics.Linecast(currentPathPoint, newPathPoint, out hit))
    {
        Vector3 direction = (currentPathPoint - hit.point).normalized;

        //Distance in front of the detected obstacle.
        float offsetDistance = 1.5f;

        if ((fishingLayer & (1 << hit.collider.gameObject.layer)) != 0)
        {
            offsetDistance = 1f;
        }

        Vector3 offsetPoint = hit.point + direction * offsetDistance;
        return offsetPoint;
    }

    return newPathPoint;
}
```

offsetDistance specifies the distance from the obstacle.

CheckPointVisibility method verifies that there are no obstacles between the newly generated point and the previous point. This check is performed to prevent potential penetration of the fishing float through obstacles during a change of direction.

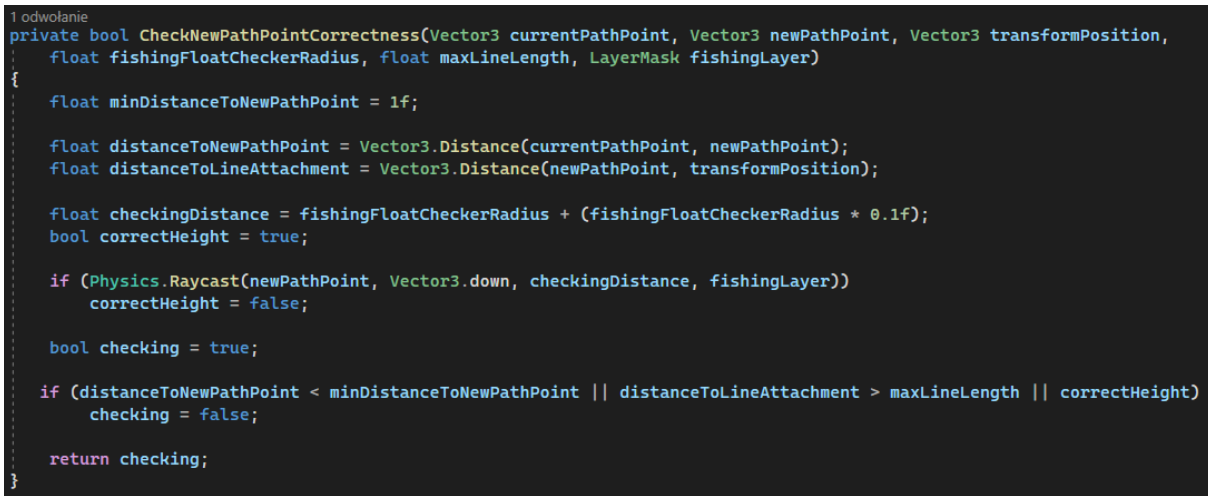


```
1 odwołanie
private bool CheckPointVisibility(Vector3 previousPathPoint, Vector3 newPathPoint, LayerMask fishingLayer)
{
    if (Physics.Linecast(previousPathPoint, newPathPoint, ~fishingLayer))
        return false;

    return true;
}
```

CheckNewPathPointCorrectness method is responsible for checking additional factors, such as:

- The distance between the newly generated point and the previous one.
- Ensuring that the newly generated point is not too far from the player.
- Verifying that the newly generated point is not in mid-air.

These checks are essential to ensure that the newly created path point adheres to specific criteria and is suitable for the fishing simulation.

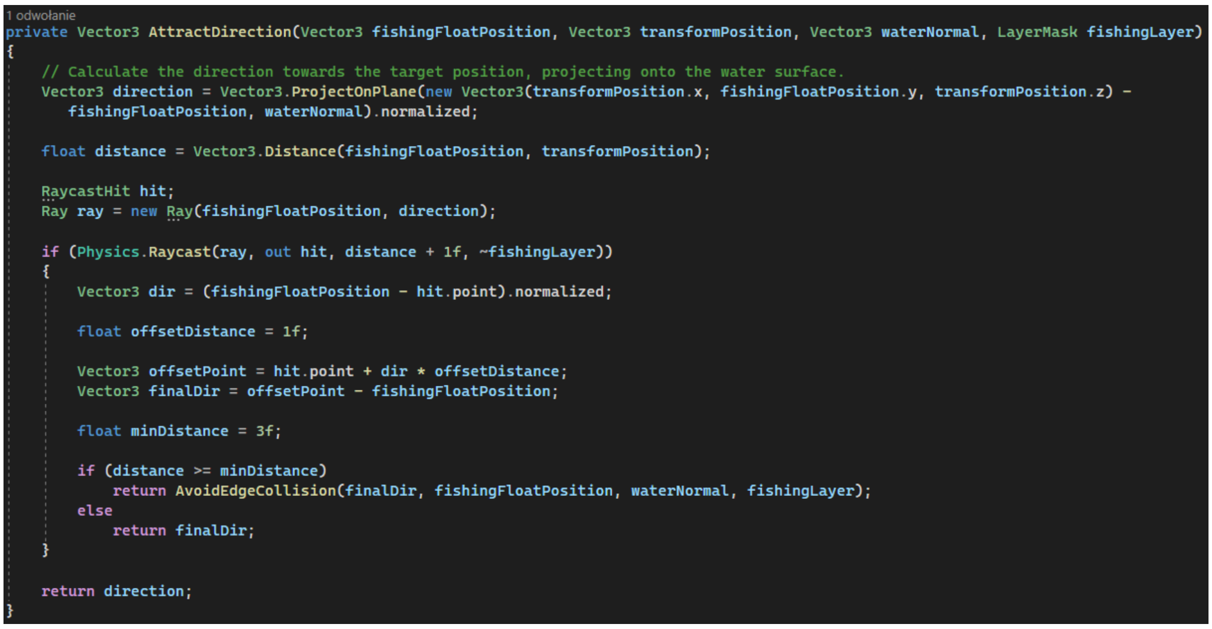


```
1 odwołanie
private bool CheckNewPathPointCorrectness(Vector3 currentPathPoint, Vector3 newPathPoint, Vector3 transformPosition,
    float fishingFloatCheckerRadius, float maxLineLength, LayerMask fishingLayer)
{
    float minDistanceToNewPathPoint = 1f;

    float distanceToNewPathPoint = Vector3.Distance(currentPathPoint, newPathPoint);
    float distanceToLineAttachment = Vector3.Distance(newPathPoint, transformPosition);

    float checkingDistance = fishingFloatCheckerRadius + (fishingFloatCheckerRadius * 0.1f);
    bool correctHeight = true;

    if (Physics.Raycast(newPathPoint, Vector3.down, checkingDistance, fishingLayer))
        correctHeight = false;

    bool checking = true;

   if (distanceToNewPathPoint < minDistanceToNewPathPoint || distanceToLineAttachment > maxLineLength || correctHeight)
        checking = false;

    return checking;
}
```

FISHING GAME TOOL

Attract Direction And Avoid Edge Collision

AttractDirection method is used to calculate the direction of attraction for the loot. It also checks for obstacles along the attraction path. If the distance between the fishing float and the player is too great, it utilizes **AvoidEdgeCollision** to circumvent obstacles.

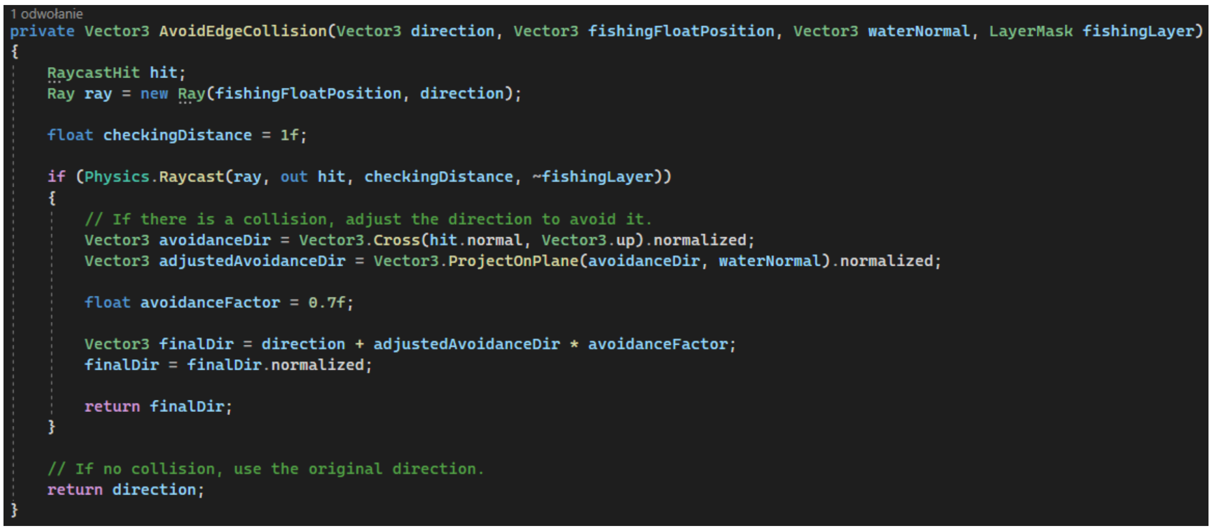


```
1 odwołanie
private Vector3 AttractDirection(Vector3 fishingFloatPosition, Vector3 transformPosition, Vector3 waterNormal, LayerMask fishingLayer)
{
    // Calculate the direction towards the target position, projecting onto the water surface.
    Vector3 direction = Vector3.ProjectOnPlane(new Vector3(transformPosition.x, fishingFloatPosition.y, transformPosition.z) -
       fishingFloatPosition, waterNormal).normalized;

    float distance = Vector3.Distance(fishingFloatPosition, transformPosition);

    RaycastHit hit;
    Ray ray = new Ray(fishingFloatPosition, direction);

    if (Physics.Raycast(ray, out hit, distance + 1f, ~fishingLayer))
    {
        Vector3 dir = (fishingFloatPosition - hit.point).normalized;

        float offsetDistance = 1f;

        Vector3 offsetPoint = hit.point + dir * offsetDistance;
        Vector3 finalDir = offsetPoint - fishingFloatPosition;

        float minDistance = 3f;

        if (distance >= minDistance)
            return AvoidEdgeCollision(finalDir, fishingFloatPosition, waterNormal, fishingLayer);
        else
            return finalDir;
    }

    return direction;
}
```

offsetDistance specifies the distance from the obstacle.

minDistance represents the minimum distance between the fishing float and the player. If the distance between them falls below this threshold, the **AvoidEdgeCollision** method ceases to be used.

AvoidEdgeCollision method is responsible for avoiding obstacles during the attraction process.



```
1 odwołanie
private Vector3 AvoidEdgeCollision(Vector3 direction, Vector3 fishingFloatPosition, Vector3 waterNormal, LayerMask fishingLayer)
{
    RaycastHit hit;
    Ray ray = new Ray(fishingFloatPosition, direction);

    float checkingDistance = 1f;

    if (Physics.Raycast(ray, out hit, checkingDistance, ~fishingLayer))
    {
        // If there is a collision, adjust the direction to avoid it.
        Vector3 avoidanceDir = Vector3.Cross(hit.normal, Vector3.up).normalized;
        Vector3 adjustedAvoidanceDir = Vector3.ProjectOnPlane(avoidanceDir, waterNormal).normalized;

        float avoidanceFactor = 0.7f;

        Vector3 finalDir = direction + adjustedAvoidanceDir * avoidanceFactor;
        finalDir = finalDir.normalized;

        return finalDir;
    }

    // If no collision, use the original direction.
    return direction;
}
```

checkingDistance specifies the checking distance of the obstacle.

avoidanceFactor is a collision avoidance factor that influences how much the direction is altered to avoid collisions.