Genetische KI

- By:
- Lukas Momberg 11141259
- Jona Siebel 11141394
- Dennis Goßler 11140150
- Thomas Donst 11138843
- Patrick Schuster 11126452

Neues Gen

- Als neues Gen wurde ein stop gen eingeführt
- Hierbei wird die Bewegung auf null gesetzt

```
public void Execute()
{
    controller.ApplyMotorTorque(0);
}
```

```
private List<char> _geneIDList = new List<char>();


public float ForwardChangsePercantage { get; set; } = 0.45f;

public void AssignGene(char geneID)
{
    _geneIDList.Add(geneID);
}

public List<Individual> CreateInitialGeneration(int generationSize, int individualSize)
{
    List<Individual> individualList = new List<Individual>(generationSize);
    char[] generatedSequence = new char[individualSize];

    for (int index = 0; index < generationSize; index++)
    {
        Individual individual = new Individual();

        for (int i = 0; i < individualSize -1; i++)
        {
            bool forward = Random.value < ForwardChangsePercantage;
            char gene = forward  ? 'B' : 'C'; // A=Reverse, B=Forward, C=Left, D=Right, E=

            generatedSequence[i] = gene;
        }

        generatedSequence[individualSize-1] = 'F';

        individual.GeneSequence = new string(generatedSequence);

        individualList.Add(individual);
    }

    return individualList;
```

Initialisierer

- Neue Liste wird erzeugt um die gesammten Checks loszuwerden
- Am Anfang wird der Agent schon mal in die richtige Richtung geschubst, man kann ja schon mal nen Hint geben wo er hin soll
- Am Ende der Sequence wird das neue gen eingeführt um ein anhalten hinzukriegen.
- Rest wird aufgefüllt.

Fitnessfunktion

- Bestraft Collision
- Nach Collision liegt der Hauptfokus auf der Distanz zum Optimum, gefolgt vom unwichtigeren Winkel
- Klare Gewichtungen oben

```
public float DetermineFitness(CarState state)
{
    const int distanceMultiplicator = 450;
    const int angleMultiplicator = 550;

    float fitness = Evaluate(state.DistanceFromGoal(), distanceMultiplicator) +
                    Evaluate(state.AngleToGoal(), angleMultiplicator);

    return (state.NumberOfCollisions() >= 1) ?
        0f : NormalizeValue(fitness, -1000, 1000);
}

private float Evaluate(float value, int multiplicator)
{
    return (1 - Mathf.Abs(value)) * multiplicator;
}

private float NormalizeValue(float value, float minValue, float maxValue)
{
    return (value - minValue) / (maxValue - minValue);
}
```

Mutator

- Mutator lässt letztes 5tel in Ruhe um das ende nicht zu verhunzen
- Maximal 4 Mutationen

```
public string Mutate(string original)
{
    var tempString = original;

    for (int i = 0; i < randomizer.Next(4); i++)
    {
        var mutatePosition = randomizer.Next(original.Length - (original.Length / 5));

        tempString = tempString.Remove(mutatePosition, 1).Insert(mutatePosition, GetRandomGene());
    }

    return tempString;
}



public string GetRandomGene()
{
    switch (randomizer.Next(5))
    {
        case 0:
            return "A";
        case 1:
            return "B";
        case 2:
            return "C";
        case 3:
            return "D";
        case 4:
            return "E";
    }
```

Crossover Recombinator

- Random Schnittpunkt
- Einfach zu verstehen
- Kann potentiel auch lediglich eine Stelle austauschen
- Oder Alles außer einer Stelle
- Tauscht ganze Blöcke

```
public class RecombinerCrossoverDonst : IRecombiner
{
    public string Combine(string parentA, string parentB)
    {
        Random random = new Random();
        int length = parentA.Length;
        int position = random.Next(0, length);

        return parentA.Substring(0, position) +
               parentB.Substring(position, length - position);
    }
}
```

Weave Recombinator

- Nimmt sich abwechselnd von einem der Parent Strings
- Gleichmäßige mischung aus beiden
- Kann bereits gute Abläufe durcheinander bringen

```
public static string CombineSequence(string parentA, string parentB)
{
    int length = parentA.Length;
    char[] equence = new char[length];

    for (int i = 0; i < length; i++)
    {
        equence[i] = ((i + 1) % 2) == 0 ? parentA[i] : parentB[i];
    }

    return new string(equence);
}
```

Selector

```
public float FitnessThreshhold { get; set; } = 0.66f;

public List<string> SelectFromGeneration(GenerationDB.Generation parentGeneration)
{
    int amountOfIndividuals = parentGeneration.individuals.Count;
    int amountOfIndividualsToTake = (int)Math.Floor(amountOfIndividuals * FitnessThreshhold);
    int amountOfIndividualsMissing = amountOfIndividuals - amountOfIndividualsToTake;
    List<string> nextGeneration = new List<string>(amountOfIndividualsToTake);

    parentGeneration.Sort();

    for (int i = 0; i < amountOfIndividualsToTake; i++)
    {
        Individual individual = parentGeneration.Individuals[i];

        nextGeneration.Add(individual.GeneSequence);
    }

    for (int index = 0; (index < amountOfIndividuals) && (amountOfIndividualsMissing-- > 0); )
    {
        Individual individualA = parentGeneration.Individuals[index++];
        Individual individualB = parentGeneration.Individuals[index++];

        string mergedSequence = RecombinerWeaveBitPaw.CombineSequence(individualA.GeneSequence, individualB.GeneSequence);

        nextGeneration.Add(mergedSequence);
    }

    if(nextGeneration.Count != amountOfIndividuals)
    {
        Debug.LogError("[SelectorDecisionBitPaw] Next generation has fewer elements than parent!");
    }

    return nextGeneration;
```

Nimmt sich das beste 3tel oder was auch immer bei FitnessThreshold definiert wird

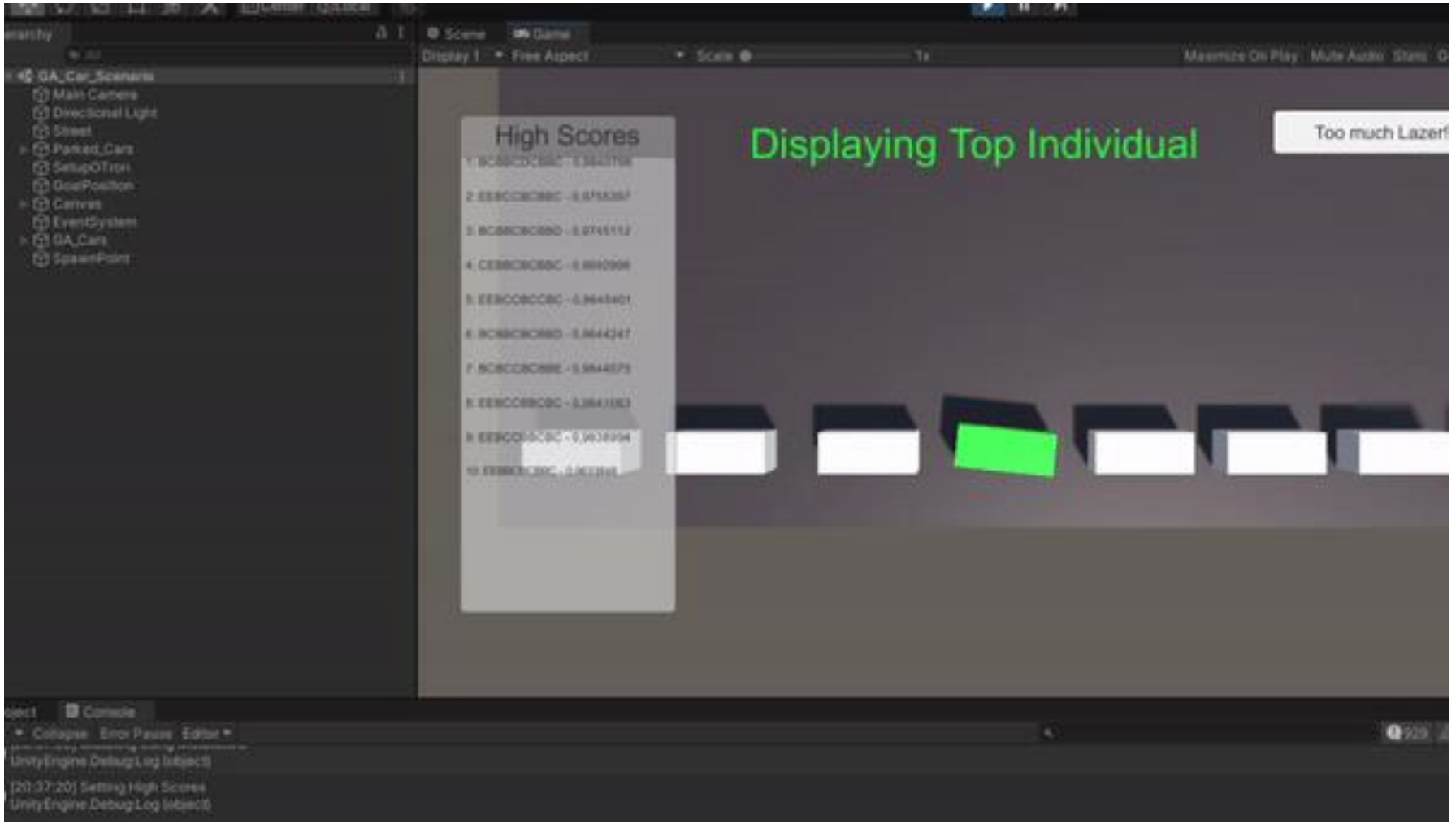
Füllt die Nächste Generation mithilfe des WeaveRecombinators und den Individuen aus der parent generation auf

Terminator

- Stoppt das Ganze, wenn der Fitness wert 0.9 überschreitet

```
public bool JudgementDay(GenerationDB.Generation generation)
{
    return generation.Fittest.Fitness >= 0.9f;
}
```

Das Ergebnis



Wir Danken für Ihre Aufmerksamkeit.