The Walker Report

Farmaki Athanasia
Student ID: 931130-T809
Email: farmaki@kth.se

August 21, 2024

1 Project Specification

In this project the aim is to implement a walking NPC (non playable character) using procedural animation. We want our NPC to look realistic and to walk around the field without bumping into obstacles. We would like whenever our character approaches an object to gracefully turn towards a feasible direction. The end goal would be to be able to insert many NPCs in a city field and watch them gracefully walk the streets and either avoid in a natural way or interact with each other. For the scope of this project we will focus on creating a walking character with some obstacle avoidance logic.

2 Theoretical Background

There are many ways to create the walking movement of an NPC. Some are the following:

1. Record animation for each possible movement and use blend trees to code the transition between them.
2. Use inverse kinematics to code a walk and, based on mathematical formulas, define ways in which the character moves the target parts.
3. Motion capture data: Using sensors, record a walking cycle and apply these animation to the characters.
4. Machine learning - reinforcement learning: Create models that are retrained during runtime based on environmental input and "learn" walking.

For the obstacle avoidance some known methods are:

1. Raycasting: Use invisible lines (rays) projected from the NPC to detect obstacles ahead. If a ray detects an obstacle, adjust the NPC's path or orientation to avoid it. [3]
2. Navigation Meshes: Divide the environment into walkable and non-walkable areas using a mesh overlay. [1]
3. Dynamic Obstacle Avoidance: Adjust paths dynamically in response to moving obstacles or other NPCs. [2]

3 Solution Description

In this project I am going to create 3 states for the character. The possible states are walk, turn and stop. I will implement the walk and the turning states using inverse kinematics and I will use a recorded animation for the idle state. More specifically for the walk I will procedurally generate a walking circle moving the legs and arms of the character to resemble a walk. Then I will program the turning of the character in a way that looks realistic and during the stop state I will use a recorded animation that I 've downloaded from mixamo

4 Implementation

4.1 Task 1: Setup the project

Description: We create a project named Walker and install the Animation Rigging package. We download a character named kaya.fbx [7] and we copy paste it inside the Assets folder. In the materials tab of the kaya.fbx we extract the textures. We also create a plane with a box collider component and we use for it the sand material from the package 4k-tiled-ground-textures [4]. The result is the following:

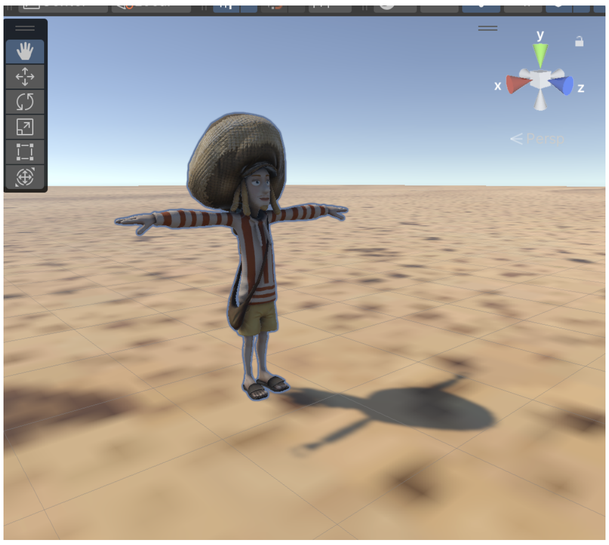


Figure 1: View after setting up the project

4.2 Task 2: Create the walking circle

Description: The objective of this task is to create a realistic walking circle. To create a basic walk we will use the animation rigging package along with inverse kinematics for the constraints. Using inverse kinematics we will be able to move the tip of the constraint and watch the rest of the bones that are connected to it move in a realistic manner. To achieve this we create a Rig object that will be added to our character. This includes all the movement constraints which in our case will include hand bones and leg bones constraints and we will use the "Two Bone IK Constraint". After creating the constraints we have to assign a target object to the "Target" field of each constraint. Each constraint will have it's own target. This will result in the tip of the constraint to always follow the target whenever the target moves.

4.2.1 Task 2.1: Create the target objects

Description: A target object is an object that the edge of a bone will follow. Using inverse kinematics the rest of the bones connected with that edge will move accordingly. To enable movement we will thus create four target spheres. One for left leg, one for the right leg, one for the left arm and one for the right arm. We will also change the position of the arms to a more natural position than the cross.

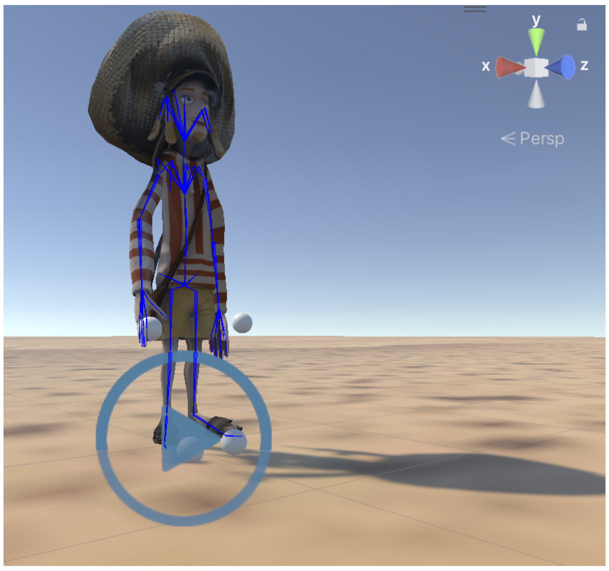
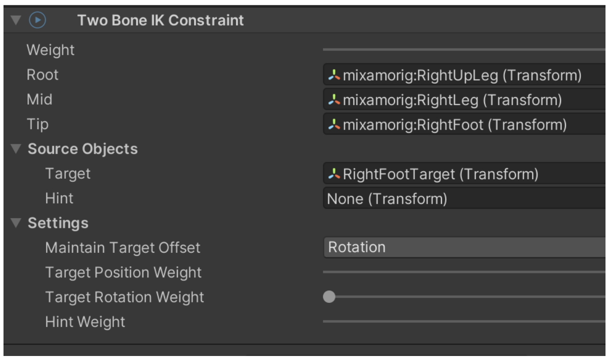


Figure 2: Target spheres

4.2.2 Task 2.1: Create the constraints

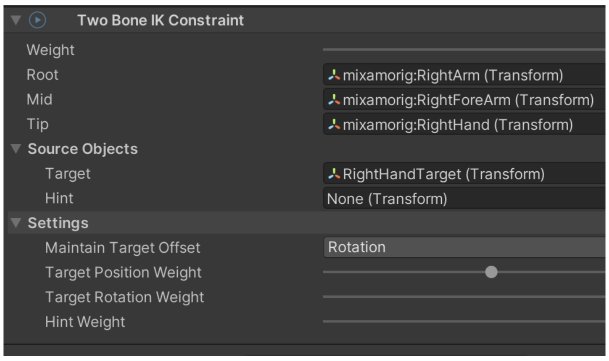
Description: In order to create the constraints we first create a empty object named WalkingRig and we add the component Rig. Then we create a child for each constraint and we name them RightLegConstraint, LeftLegContraint, RightHandConstraint, LeftHandConstraint. We need to assign bone objects to the fields Root, Mid, Tip and we also have to assign a Target. Some constraint configurations are the following:



Two Bone IK Constraint

Weight	
Root	mixamorig:RightUpLeg (Transform)
Mid	mixamorig:RightLeg (Transform)
Tip	mixamorig:RightFoot (Transform)
Source Objects	
Target	RightFootTarget (Transform)
Hint	None (Transform)
Settings	
Maintain Target Offset	Rotation
Target Position Weight	
Target Rotation Weight	
Hint Weight	

Figure 3: Right leg constraint


Two Bone IK Constraint

Weight	
Root	mixamorig:RightArm (Transform)
Mid	mixamorig:RightForeArm (Transform)
Tip	mixamorig:RightHand (Transform)
Source Objects	
Target	RightHandTarget (Transform)
Hint	None (Transform)
Settings	
Maintain Target Offset	Rotation
Target Position Weight	
Target Rotation Weight	
Hint Weight	

Figure 4: Right hand constraint

The result is that by moving the targets we observe the limbs move accordingly.

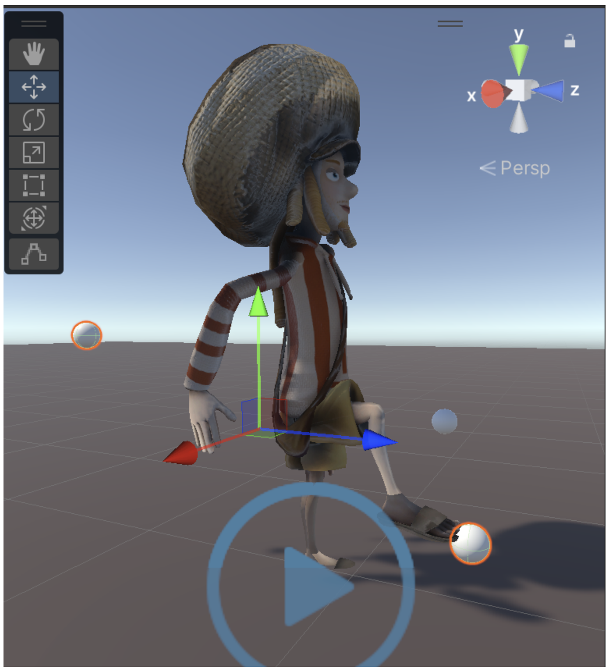


Figure 5: Task 2 result

4.2.3 Task 2.3: Create a walking script

Description: Create a script that moves the targets using mathematical formulas in the form of a curve.

Create target variables:

In the script we first create public Vector3 variables for the target objects. This is were we will assign in unity our target spheres. We also create private variables for the target' offsets. In the Start function of the Monobehaviour class we initialize the offset to be the local position of the target. This is very important since if we don't do that and start moving the target any local position change will be relevant to the parent of the target object which is either (0,0,0) if it has no parent or the position of the character if we set the targets to be children of the character.

Here are the code snippets described above:

```

```

```
    // In these variables we assign the target spheres
    public Transform leftFootTarget;
    public Transform rightFootTarget;
    public Transform leftHandTarget;
    public Transform rightHandTarget;

    private Vector3 leftFootTargetOffset;
    private Vector3 rightFootTargetOffset;
    private Vector3 leftHandTargetOffset;
    private Vector3 rightHandTargetOffset;

    void Start()
    {
        leftFootTargetOffset = leftFootTarget.localPosition;
        rightFootTargetOffset = rightFootTarget.localPosition;

        leftHandTargetOffset = leftHandTarget.localPosition;
        rightHandTargetOffset = rightHandTarget.localPosition;
    }


    void SetTargetPosition(Transform target, Vector3 offset,
       float forward, float upwards){

        // Performs the mathematical operations to change the
           target position using the  arguments forward and
           upwards that determine the change we wish to apply on
           the forward direction and the up direction


        // InverseTransformVector transforms a vector from world
           space to local space relative to 'this.transform'
           space
        Vector3 moveForward = this.transform.
           InverseTransformVector(target.forward) * forward;
        Vector3 moveUp = this.transform.InverseTransformVector(
           target.up) * upwards;

        // set the local position of the traget by adding the
           above vectors
        target.localPosition = offset + moveForward + moveUp;
    }
```

The result is that by moving the target spheres we observe the limbs move accordingly.

Setup the Animation Curves:

We need some mathematical formulas to determine how each target will move in what we will determine as one period of time. Unity provides us with an object type called AnimationCurve that can do exactly that. We will tweak the parameters in order to find the ones that make our walk convincing. We first create four public AnimationCurve variables. One for the forward direction of the legs, one for the forward direction of the arms, one for the upward direction of the legs and one for the arms. We will not move our targets at all in the right/left direction.

```
    public AnimationCurve legHorizontalCurve;
    public AnimationCurve legVerticalCurve;
    public AnimationCurve armHorizontalCurve;
    public AnimationCurve armVerticalCurve;
```

After significant experimentation I ended up choosing these curves. For the forward direction of the legs (**range: -1 to 0.5**) :

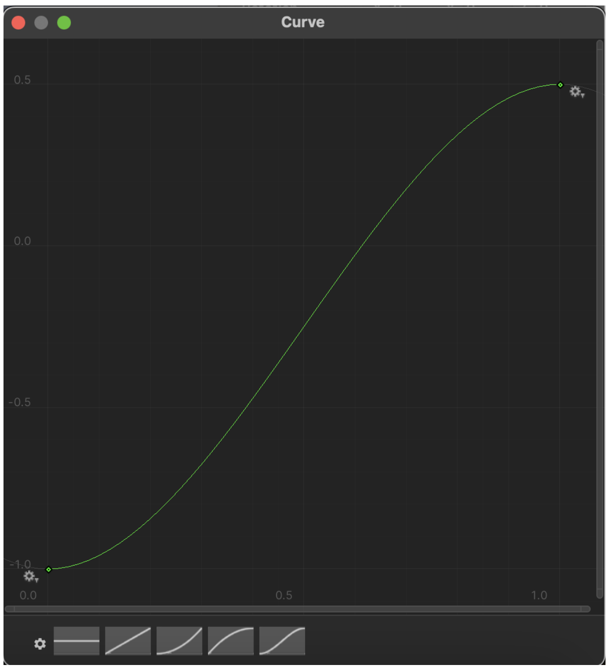


Figure 6: Leg horizontal curve

For the vertical direction of the legs (**range: 0 to 1**):

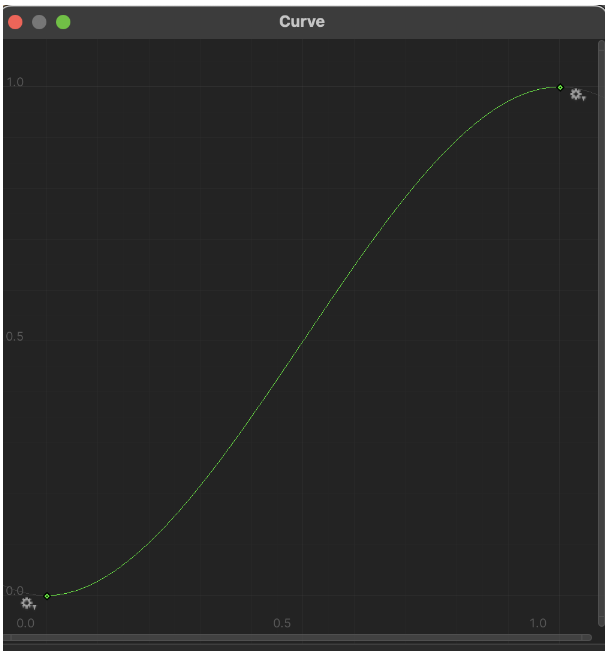


Figure 7: Leg vertical curve

For the forward direction of the arms (**range: 0 to 1**) :

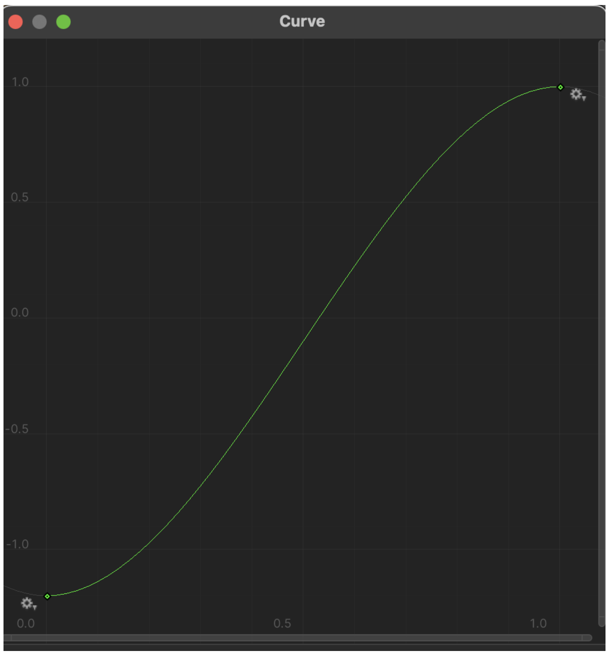


Figure 8: Arm horizontal curve

For the vertical direction of the arms (**range: 0 to 1**):

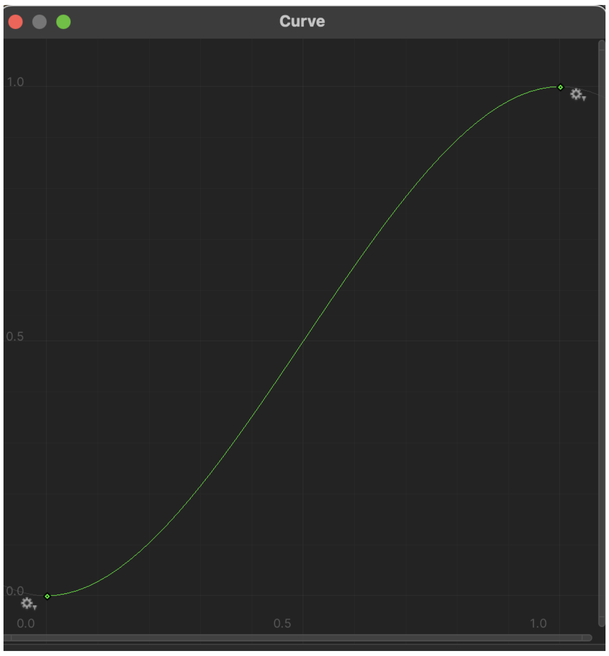


Figure 9: Arms vertical curve

We also select ping pong in the curves settings:

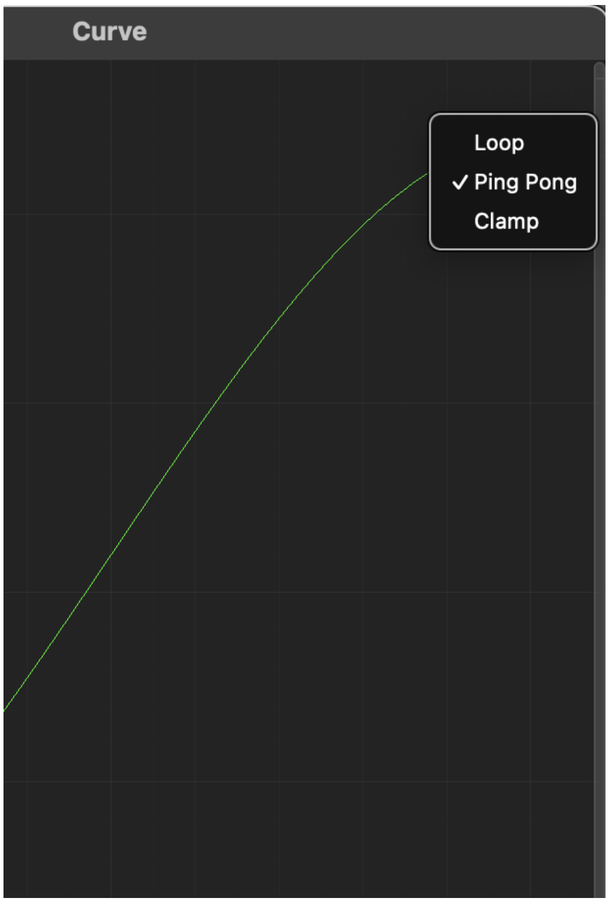


Figure 10: Ping pong

In order to move to move the targets we create four functions:

1. moveLeftFootTarget
2. moveRightFootTarget
3. moveLeftHandTarget
4. moveRightHandTarget

Lets analyse one of them.

```
    // frequency sets the time of a period. Affects the speed of movements.
    float adjustedTime = Time.time * frequency;


    float moveLeftFootTarget(float adjustedTime){
        float forward = legHorizontalCurve.Evaluate(adjustedTime) * 0.3f;
        float upward = legVerticalCurve.Evaluate(adjustedTime+0.5f) * 0.2f;
        SetTargetPosition(leftFootTarget,  leftFootTargetOffset, forward, upward);
        float forwardDirection = forward - leftFootLastForwardMovement;
        leftFootLastForwardMovement = forward;
        return forwardDirection;
    }

    void SetTargetPosition(Transform target, Vector3 offset, float forward, float upwards){

        // Performs the mathematical operations to change the target position using the  arguments forward and upwards that determine the change we wish to apply on the forward direction and the up direction


        // InverseTransformVector transforms a vector from world space to local space relative to 'this.transform' space
        Vector3 moveForward = this.transform.InverseTransformVector(target.forward) * forward;
        Vector3 moveUp = this.transform.InverseTransformVector(target.up) * upwards;

        // set the local position of the traget by adding the above vectors
        target.localPosition = offset + moveForward + moveUp;
    }
```

Adjusted time: Instead of having one period lasting one second we can multiply it by the number we wish to consider as period and we call this adjusted time. The default in our implementation is 1.5 seconds but the field frequency is public and we can change it

is unity.
line 6: Evaluate the legHorizontalCurve on the adjustedTime to get how the forward position of the target should change for that particular time. We scale it by 0.3 cause we want a smaller movement of the leg forward and backward.
line 7: Evaluate the legVerticalCurve on the adjustedTime to get how the upward position of the target should change for that particular time. We scale it by 0.3 cause we want a smaller movement upward than the curve indicates.
line 8: Call the SetTargetPosition function which is responsible for actually changing the position of the target
line 9,10,11: In these lines we keep track of the amount of change in the position in each direction in the **current frame** and we will use this to change the character's world position so that the character actually moves as much as the legs move. Note that the character will move only when the change is negative by the amount of calculated here.

moveRightFootTarget function:

```
    float moveRightFootTarget(float adjustedTime){
        float forward = legHorizontalCurve.Evaluate(adjustedTime-1) * 0.3f;
        float upward = legVerticalCurve.Evaluate(adjustedTime-0.5f) * 0.2f;
        SetTargetPosition(rightFootTarget,  rightFootTargetOffset, forward, upward);
        //  The forwardDirection variable is how much the character position should change in the forward direction.
        // It's the current position amount of change - the change in the previous frame
        // In our character moving logic we will move the character only when this is negative.
        float forwardDirection = forward - rightFootLastForwardMovement;

        // rightFootLastForwardMovement  is used to keep track of the last forward movement change.
        rightFootLastForwardMovement = forward;
        return forwardDirection;
    }
```

This function is almost the same as the moveLeftFootTarget that we analysed above. The differences are that we evaluate the horizontal curve on adjustedTime-1 as we want the left leg to be behind when the right is forward. Also the rising of the leg happens in time adjustedTime-0.5f as it is when the leg is exactly where the body is (not forward and not backward).

moveLeftHandTarget function:

```
    void moveLeftHandTarget(float adjustedTime){
        float forward = armHorizontalCurve.Evaluate(adjustedTime-1f) * 0.4f;
        float upward = armVerticalCurve.Evaluate(adjustedTime) * 0.01f;
        SetTargetPosition(leftHandTarget, leftHandTargetOffset,
           forward, upward);
    }
```

This function is also similar to the moveLeftFootTarget. Here we use a different curve and we evaluate it on time adjustedTime-1f so that the left arm is forward when the right leg is forward. Also we use different scalars here as these created a more natural walk.

moveRightHandTarget function:

```
    void moveRightHandTarget(float adjustedTime){
        float forward = armHorizontalCurve.Evaluate(adjustedTime)
            * 0.4f;
        float upward = armVerticalCurve.Evaluate(adjustedTime) *
           0.01f;
        SetTargetPosition(rightHandTarget, rightHandTargetOffset
           , forward, upward);
    }
```

Here we use the same curves as the left hand but we evaluate it on time adjustedTime so that the right arm is moving opposite to the left hand when the right leg is forward.

The result of the script is the following video

Walk function:
Last but not least we add the actual movement of the character. For now we will assign some keyboard controls to activate each state to be able to test the system. The walking cycle is the state: Walking. In the following sections we will implement a turning state and a stop/idle state.

```

    public enum State{
        Walking,
        Turning,
        Stop
    }

    void Update()
    {
        float adjustedTime = Time.time * frequency;
        if (Input.GetKeyDown(KeyCode.UpArrow)){
            activeState = State.Walking;
        }

        if (activeState == State.Walking){
            Walk(adjustedTime);
        }
        else if ( activeState == State.Turning){
            Walk(adjustedTime);
            Turn();
        }
        else{
            Stop();
        }
    }

    public void Walk(float adjustedTime){
        // Walking movement
        // move the target that the left foot tip is following
        float  leftLegDirectionforward  = moveLeftFootTarget(
           adjustedTime );
         // move the target that the right foot tip is following
        float rightLegDirectionforward = moveRightFootTarget(
           adjustedTime);
        // move the target that the left hand is following
        moveLeftHandTarget(adjustedTime);
        // move the target that the right hand is following
        moveRightHandTarget(adjustedTime);

        // move game object forward when the foot hits the floor
        RaycastHit hit;
        // find the position where a vertical line starting from
           the target hits the floor
        bool raycastHittingFloor = Physics.Raycast(leftFootTarget
           .position + leftFootTarget.up, -leftFootTarget.up, out
            hit, 10f );
        // if the leg moves backwards set the target to be on the
            floor for aesthetical reasons
        // Also when the leg moves backward move the character
        if ( leftLegDirectionforward<0 && raycastHittingFloor ){
            leftFootTarget.position = hit.point;
            this.transform.position += this.transform.forward *
               Math.Max(-leftLegDirectionforward, 0F);
        }

        // Same logic as in the code snippet above but for the
           right leg
        raycastHittingFloor = Physics.Raycast(rightFootTarget.
           position + rightFootTarget.up, -rightFootTarget.up,
           out hit,  10f);
        if ( rightLegDirectionforward<0 && raycastHittingFloor ){
            rightFootTarget.position = hit.point;
            this.transform.position += this.transform.forward *
               Math.Max(-rightLegDirectionforward, 0f);
        }
    }
```

We first call the moving functions mentioned above and then we add the if condition which if the leg moves backwards (rightLegDirectionforward¡0:) and the Raycast hits the ground in some position:

1. Set the foot position to the position where the raycast hits the ground which makes the walk more realistic.

2. move the character's world position by the ammount of change that we explained

above which makes the character move as much as we move the targets.

The final result of the walking circle can be found HERE

4.3 Task 3: Create the turn state

To turn our character realistically we have to change the rotation. In Unity any rotation change happens using quaternions [10]. The tricky part is to make this turn slowly and distribute it into many frames. Let's see the implementations:

```

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.LeftArrow) && !(activeState==State.Turning))
        {
            activeState = State.Turning;
            // ElapsedTime holds the time passed from the start of the turn. When reaching turnDuration we stop turning
            elapsedTime = 0.0f;
            startRotation = transform.rotation; // Important step: set the current rotation as starting
            rotationChange = Quaternion.AngleAxis(-45f, this.transform.up); // Set rotation change 45 degrees left
        }
        if (Input.GetKeyDown(KeyCode.RightArrow) && !(activeState==State.Turning))
        {
            activeState = State.Turning;
            // ElapsedTime holds the time passed from the start of the turn. When reaching turnDuration we stop turning
            elapsedTime = 0.0f;
            startRotation = transform.rotation; // Important step: set the current rotation as starting
            rotationChange = Quaternion.AngleAxis(45f, this.transform.up); // Set rotation change 45 degrees right
        }
        // frequency sets the time of a period. Affects the speed of movements.
        float adjustedTime = Time.time * frequency;
        if (activeState == State.Walking){
            Walk(adjustedTime);
        }
        else if ( activeState == State.Turning){
            Walk(adjustedTime);
            Turn();
        }
        else{
            Stop();
        }
    }
```

In the Update function for testing purposes we added a keyboard control that activates the turning. The leftarrow is used to turn the character left and the right is to turn him right 45 degrees. When turning is activated we set the active state to State.Turning and the elapsedTime variable to zero. This variable counts the time after the start of tehe turning. When turnDuration threshold is reached we stop the turning. We also calculate the startRotation and the rotationChange which are both of type Quaternion. Then Turn function is called:

```
        private IEnumerator RotateOverTime(Quaternion
           rotationChange, float duration, float elapsedTime)
    {
        while (elapsedTime < duration)
        {
            // Slerp is a spherical linear interpolatio  used to
               smoothly interpolate between the starting rotation
                and the target rotation.
            transform.rotation = Quaternion.Slerp(startRotation,
               startRotation * rotationChange, elapsedTime /
               duration);
            elapsedTime += Time.deltaTime;
            yield return null;
        }
    }


    void Turn(int? deg, Quaternion rotationChange){
        // Coroutines are able to distribute  the execution into
           many frames
        float turnDuration = Math.Max(Math.Abs((float)Math.Max((
           float) deg/20.0, 1.0)), 3f);

        StartCoroutine(RotateOverTime(rotationChange,
           turnDuration, elapsedTime));
        elapsedTime += Time.deltaTime;
        if (elapsedTime>turnDuration){
            unobstractedDeg = new List<int>();
            turningDeg=0;
            activeState = State.Walking;
            elapsedTime=0f;
            rotationChange = new Quaternion(0f, 0f, 0f, 0f );
        }
    }
```

The turn is calling a coroutine [8] that uses the Quaternion Slerp to turn the character from the starting rotation to the end in a given duration over multiple frames. In Unity, a coroutine is a method that can pause its execution (yield) and then continue from where it left off in the following frame. The result of this implementation can be seen in this video.

4.4 Task 4: Create the stop state

To create the stop state that sets the character into an idle status we used a recorded animation. We will download an animation [9] from mixamo. Then in order to intergrate this animation with our character we do the following:

1. Add the .fbx animation on Assets
2. Create an animations folder and add a copy of the animation clip and an animation controler

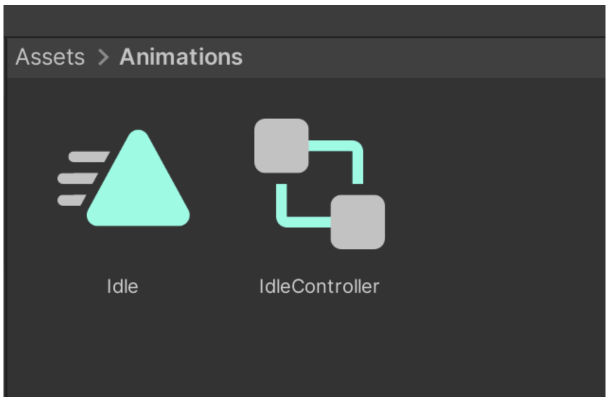
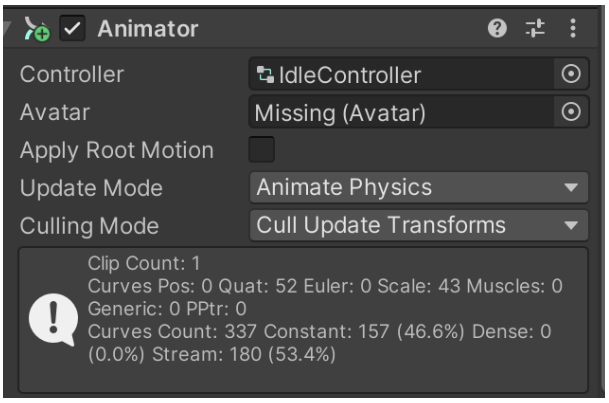


Figure 11: Animations folder

3. Add the animation clip in animation controller and assign the animation controller on the field controller of the animation component of the character



Animator

Controller: IdleController
Avatar: Missing (Avatar)
Apply Root Motion: ☐
Update Mode: Animate Physics
Culling Mode: Cull Update Transforms

```
Clip Count: 1
Curves Pos: 0 Quat: 52 Euler: 0 Scale: 43 Muscles: 0
Generic: 0 PPtr: 0
Curves Count: 337 Constant: 157 (46.6%) Dense: 0
(0.0%) Stream: 180 (53.4%)
```

Figure 12: Animation controller

4. Add a bool parameter on the animator controller that will activate and deactivate the animation. That in my case is called IsIdle

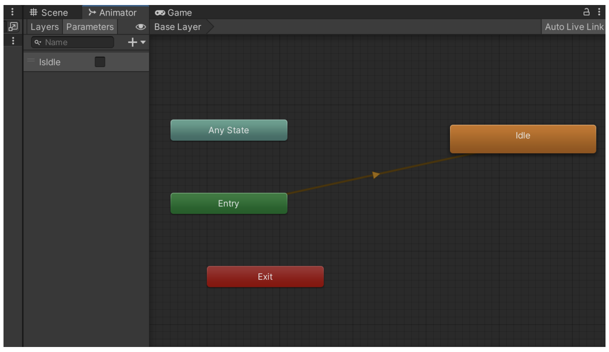
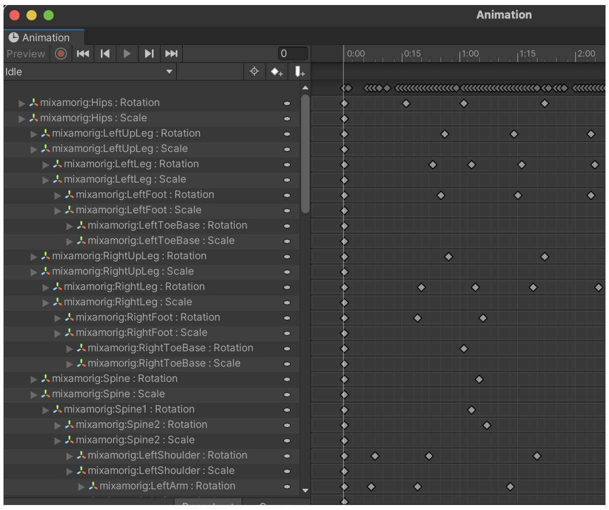


Figure 13: IsIdle variable

Issues:

After adding the animation to my character the character's position changed. To solve the issue i opened the animation clip and removed the position property.



```
mixamorig:Hips : Rotation
mixamorig:Hips : Scale
  mixamorig:LeftUpLeg : Rotation
  mixamorig:LeftUpLeg : Scale
    mixamorig:LeftLeg : Rotation
    mixamorig:LeftLeg : Scale
      mixamorig:LeftFoot : Rotation
      mixamorig:LeftFoot : Scale
        mixamorig:LeftToeBase : Rotation
        mixamorig:LeftToeBase : Scale
  mixamorig:RightUpLeg : Rotation
  mixamorig:RightUpLeg : Scale
    mixamorig:RightLeg : Rotation
    mixamorig:RightLeg : Scale
      mixamorig:RightFoot : Rotation
      mixamorig:RightFoot : Scale
        mixamorig:RightToeBase : Rotation
        mixamorig:RightToeBase : Scale
  mixamorig:Spine : Rotation
  mixamorig:Spine : Scale
    mixamorig:Spine1 : Rotation
      mixamorig:Spine2 : Rotation
      mixamorig:Spine2 : Scale
        mixamorig:LeftShoulder : Rotation
        mixamorig:LeftShoulder : Scale
          mixamorig:LeftArm : Rotation
```

Figure 14: Remove position

Unfortunately, there is one more issue which is that the use of isIdle variable to activate and deactivate the idle animation doesn't work. I will deactivate the animation and add the fixing of the deactivation logic as future work but we can see that mixing procedural animation and pre recorded ones can be tricky.

4.5 Task 5: Add obstacles and setup camera

Description: Add obstacles in the field in order to implement the events that will activate the states transition. We created a small field bounded by 4 walls and then three cylinders and a cube was introduced. We assign two materials one for the walls and one for the obstacles that have a standard shader and the default settings. We only change the colors.

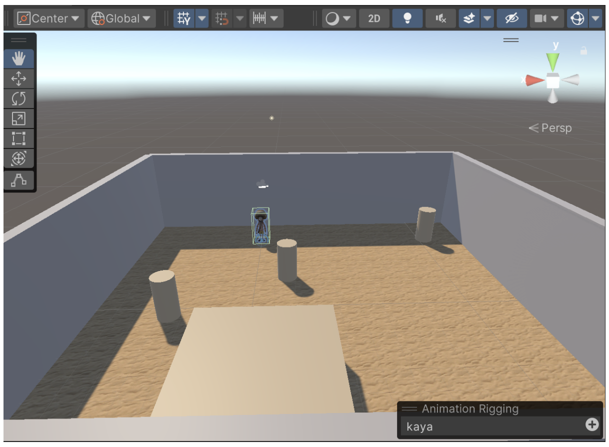


Figure 15: Field picture

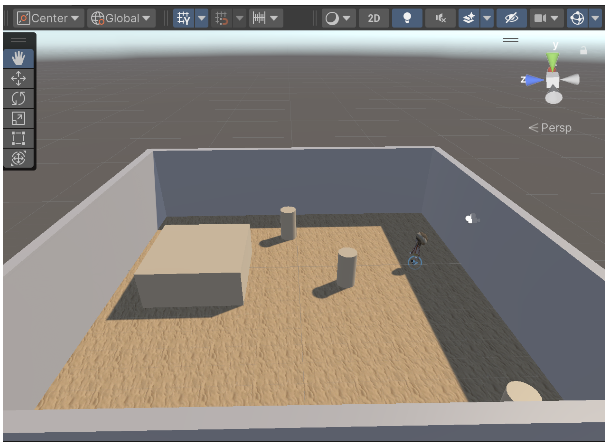


Figure 16: Field picture

To setup the camera we have to fix the initial position and also to create a script that controls the camera position as our NPC moves around.

4.5.1 Camera initial position:

Navigate with the mouse on the desired position in our case behind the character and "Align With View".

4.5.2 Camera runtime position:

```
using UnityEngine;

public class CameraFollowZ : MonoBehaviour
{
    // The transform of the player to follow
    public Transform playerTransform;

    // The offset on the Z-axis for the camera
    public float zOffset = -0.2f;
    public float yOffset = 0f;

    // Initial position of the camera
    private Vector3 initialPosition;

    void Start()
    {
        // Store the initial position of the camera
        initialPosition = transform.position;
    }

    void LateUpdate()
    {
        // Create a new position for the camera that only changes
            on the Z-axis
        Vector3 newPosition = new Vector3( playerTransform.
           position.x,
                                          initialPosition.y,
                                           playerTransform.
                                              position.z +
                                              zOffset);

        // Update the camera's position
        transform.position = newPosition;
    }
}
```

In the code above on Start we initialize the initialPosition with the current camera position and during the LateUpdate we update the position following the character's position on the x and the z axis. The LateUpdate method is used here because we want the camera positioning logic to execute after all Update logic has been processed.

4.6 Task 7: Implement a basic obstacle avoidance logic

Description: Last but not least we need to implement a logic with which our NPC will walk around the field without bumping into any of the obstacles or walls. To do that we will use Raycasting. The logic we implement is the following:

1. Check if there is an obstacle in front of the character by raycasting on 4 meters distance. If not keep the active status.
2. If there is an obstacle, create Raycasts on discrete values from -130 degrees to 130 degrees on 5 meters distance and keep the ones that no collision was detected on a list. The turning degrees in the range -90 to 90 have priority and only if we find no solution on that range we search the -130 to 130 degrees range.
3. Randomly pick one of the degrees and turn the character on that direction. While the character is turning no other turning command can be scheduled. Only after the current turn is over another turn can be done.

Let's take a look at some important parts of the code:

For a turn to happen we need to set the activeState to State.Turning and determine the variables turningDeg, rotationChange.

```
        if ( activeState == State.Turning){
            Walk(adjustedTime);
            Turn(turningDeg, rotationChange);
        }
```

In the following snippet we determine whether we need to turn and how many degrees.

```
        // check if there is an obstacle in front of the
           character
        Vector3 rayPosition = transform.position;
        rayPosition.y += 1.0f; // tackles a ray slope issue

        // check if there is  an obstacle in front of the
           character
        bool hitFlag0 = CheckForObstacles(rayPosition);

        // if there is and no other turn is in progress
        if (hitFlag0 && activeState != State.Turning){

            // find unobstructed turns in the range [-90, 90]
            //that the character can take
            unobstractedDeg = FindFeasibleDegrees(rayPosition,
               unobstractedDeg, false);

            // if no feasible turn is found extend range and
               search again
            if (unobstractedDeg.Count == 0){
                bool extendedSearch = true;
                unobstractedDeg = FindFeasibleDegrees(rayPosition
                   , unobstractedDeg, extendedSearch);
            }

            if (unobstractedDeg.Count > 0)
            {
                // Pick one of the turns from the list
                turningDeg = PickTurningDegree(unobstractedDeg);

                // set the variables needed to turn
                activeState = State.Turning;
                elapsedTime = 0.0f;
                startRotation = transform.rotation;
                rotationChange = Quaternion.AngleAxis((float)
                   turningDeg, this.transform.up);
            }
        }
```

The utilities functions used on the code snippet above are:

1. **CheckForObstacles**

```
    private bool CheckForObstacles(Vector3 rayPosition){
        RaycastHit hitPoint;
        bool hitFlag;

        Quaternion rotation0  = Quaternion.Euler(0, 0, 0);
        Vector3 direction0 = rotation0 * transform.forward;
        hitFlag = Physics.Raycast(rayPosition, direction0,
           out hitPoint,  4f);
        return hitFlag;
    }
```

2. **FindFeasibleDegrees**

```
    public List<int> FindFeasibleDegrees(Vector3 rayPosition,
        List<int> unobstractedDeg, bool extendedSearch){
        RaycastHit hitPoint;
        List<int> degrees;
        if (extendedSearch){
            // extended search candidate turns
            degrees =  new List<int> {-130, -120,- 100, 100,
               120, 130};
        }
        else{
            // default candidate turns
            degrees =  new List<int> {-90, -60, -45, -30,
               -20, -10, 0, 10, 20, 30, 45, 60, 90};

        }
        bool hitFlag;

        // Increment the Y position by 1 meter
        foreach (int deg in degrees){

            // We use Quaternion.AngleAxis with the up
               direction as rotation axis to get a ray that
               starts from the character and pins in front of
                him with an deg angle
            Quaternion rotation  = Quaternion.AngleAxis((
               float) deg, this.transform.up);

            // apply the rotation in relation to the
               charachers forward direction
            Vector3 direction = rotation * transform.forward;

            // check for obstacles in distance of 5 meters
            hitFlag = Physics.Raycast(rayPosition, direction,
                out hitPoint,  5f);

            // if no obstacle is found add the degree under
               investigation to the candidate turns list
            if (!hitFlag && !unobstractedDeg.Contains(deg)){
                unobstractedDeg.Add(deg);
            }
        }
        return unobstractedDeg;
    }
```

3. **PickTurningDegree**

```
    private int? PickTurningDegree(List<int> unobstractedDeg)
       {
        System.Random random = new System.Random();
        int randomIndex = random.Next(0, unobstractedDeg.
           Count);
        turningDeg = unobstractedDeg[randomIndex];
        return turningDeg;
    }
```

The result can be seen on THIS video.

5 Future Improvements

As future improvements we could:

1. Fix the deactivation of the Idle state since currently the idle state is not integrated
2. Improve the procedural walk to make it more realistic using more target spheres more constraints and experiment with different mathematical formulas that control the movement of the targets
3. Implement a more sophisticated obstacle avoidance and in general the moving around the field logic. The next step for me would be to try Machine Learning models for that.
4. Add more states with the character being able to run, jump etc.
5. Create more beautiful field that consists of a designed environment rather than a set with cubes and cylinders.

6 References to Resources

References

[1] Rabin, S. (Ed.). (2013). Chapter 27 Tactical Pathfinding on a NavMesh. In *Game AI Pro: Collected wisdom of game AI professionals*. CRC Press.

[2] Unity Technologies. *Navigation Package Documentation*. Available at: `https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/index.html`. Accessed: 2024-08-03.

[3] Unity Technologies. *Physics.Raycast*. Available at: `https://docs.unity3d.com/ScriptReference/Physics.Raycast.html`. Accessed: 2024-08-03.

[4] Unity Technologies. *4K Tiled Ground Textures - Part 1*. Available at: `https://assetstore.unity.com/packages/2d/textures-materials/4k-tiled-ground-textures-part-1-269480`. Accessed: 2024-08-03.

[5] Unity Technologies. *Unity Animation Rigging Package Manual.* Available at: `https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.0/manual/index.html`. Accessed: 2024-08-04.

[6] Adobe. *Mixamo Animation Library.* Available at: `https://www.mixamo.com/#/?page=2`. Accessed: 2024-08-04.

[7] Adobe. *Mixamo Character Library.* Available at: `https://www.mixamo.com/#/?page=1&type=Character`. Accessed: 2024-08-04.

[8] Unity Technologies. *Coroutines.* Available at: `https://docs.unity3d.com/Manual/Coroutines.html`. Accessed: 2024-08-04.

[9] Mixamo. *Orc Idle Animations.* Available at: `https://www.mixamo.com/#/?page=1&query=orc+idle&type=Motion%2CMotionPack`. Accessed: 2024-08-09.

[10] Boris the Brave. (2022). *Everything You Need to Know About Quaternions for Game Development.* Available at: `https://www.boristhebrave.com/2022/12/12/everything-you-need-to-know-about-quaternions-for-game-development/`. Accessed: 2024-08-04.