AI testbed scenario

Artificial Intelligence for Game Developers

This report is the outcome of a project set by the 'Artificial Intelligence for Game Developers' module, CS3S667. For the represented course of 'Computer Games Development'

The objective to this report is to document, evaluate and discuss improvements to the pre-built environment provided via github named: Dino-park. This is a testbed scenario set to build upon an already existing project and to show findings to positive or negative impacts to changes made.
Each change will show snippets and screenshots if necessary to back-up evidence to the results of each change.

14081032

Introduction

The assignment was to test and run a simulation of AI within a test bed scenario, a part-made state machine linked to dino's within the park "Anky" and "Rapty" were to be expanded on through the state machine provided. This was to test a real-life scenario of developers within the industry that look to work alongside companies that already have they're own base code in place and to help develop techniques that can be passed into the business through the Unity engine provided that was free to download. The dino-park was accessed via github through "Placeholder games" and changes made are to be discussed in how they were implemented and why. All changes to both the Anky and Rapty are listed below. Each implementation has a code snippet that helps explain each change and the location of that change within the C# files that are scripted into the AI testbed. After changes made an Appendix showing screenshots of the testcode in action is provided.

Anky Changes

The Anky controller had a full state machine but was empty. The first task was to initialise the States by setting a parameter in the Idle state to move into the Grazing state. A simple hunger bar can be created using a public float and passed to the Anky. This number will increase using the '+=' function at a rate determined by a public variable, allowing the rate to be adjusted more easily. Once a threshold is reached the state moves into the "Grazing" state(Fig, 1). The return condition to Idle state would be after grazing the eating state will be initialised and once a condition is met will push the transition back to Idle. The problem came when a Rapty entered the area, prioritising finishing the task before pushing into the escape state. For a future development adding this as a priority over tasks would ensure a better survival rate and would be a future development.

```
public float hunger = 5;
hunger = animator.GetFloat("hungerValue");
hunger += 0.8f;
animator.SetFloat("hungerValue", hunger);
if (hunger <= 5)
{
  anim.SetBool("isGrazing", true);
  anim.SetBool("isIdle", false);
}
```

(Figure, 1. Snippets of code showing the values created and the variables explained above.)

Another option if I had more time would be to add a herding mechanic in as a start-up option, this can be implemented by finding the locations of other Anky and if this distance between them is beyond a threshold switch to a herding state that allows the Anky to group up with similar animator groups. This would require to group the Anky animators together using an array to allow access to other Anky locations using the "Vector3" matrix to handle location data to group the Anky together and a condition that if 1 or more Anky are within a threshold to switch to Idle or Grazing state. This method would boost survivability for the Anky by offering strength in numbers. The method can be more effective if more students also did this to ensure a group of Ankys could take down a Rapty.

A Grazing state was needed for the Anky to still eat as all animals eat, this would allow the Rapty to attack the Anky if it is alone eating as do most predators attack at this most vunerable time of need, a grass object is created and linked to a hit box.
Using grazing state, a public variable Transform was used to hold the grass location data. On entry to the state the grass location is taken. Upon updates the animator (Anky) moves to the grass vector point using "vector2.MoveTowards" and setting a waypoint for the Animator(Fig, 2).

```
public Transform GrazingPos;
```

```
GrazingPos = GameObject.FindGameObjectWithTag("Grass").transform;
animator.transform.position = Vector2.MoveTowards(animator.transform.position,
GrazingPos.position, speed * Time.deltaTime);
```
(Figure, 2. example of the coding technique used to move the Anky to the grass location.)

A box collision is created in the myAnky file to accommodate for the condition met when colliding with the grass, the state is then switched to “Eating”. A simple collision box was created, linked to the grass position and when entered switched states and destroys the object(Fig, 3). For a more complex condition, adding a decaying aspect to Anky corpses and applying a time within the collision box that if within the box for longer than 3 seconds reduces the hunger by an amount and destroys the box.

```
public Transform Ankylocation;
public BoxCollider boxCol;
Ankylocation = GetComponent<Transform>();
boxCol = GetComponent<BoxCollider>();
anim.SetFloat("AnkyX", Ankylocation.position.x);
anim.SetFloat("AnkyY", Ankylocation.position.y);
anim.SetFloat("AnkyZ", Ankylocation.position.z);
if (boxCol.gameObject.tag == "GrazingPos")
{
  anim.SetBool("isEating", true);
  anim.SetBool("isGrazing", false);
}

// myAnky
if (boxCol.gameObject.tag == "GrazingPos")
{
  anim.SetBool("isEating", true);
  Debug.Log("Anky is Eating Grass");
  Destroy(boxCol.gameObject);
}
```
(Figure, 3. Snippets of the collision aspects of the Grass and initialisation of the location data for the Anky.)

The results of this was that then Anky would move to a grass location and the grass object would destroy itself(See Appendix).

More get’s and set’s are initiated within the state engine placeholder to change states when called upon to ensure the state engine does switch states when required and was used as testing/validation to the state engine being set to “MyAnky”(Fig, 4).

A key press to play the animation and allow the anky to flee was inserted into the Input.GetKeyDown function, this was to play the fleeing animation of the anky to run from the Rapty during playtime and was for testing purposes. It was also to allow the user to force the flee animation during “times of need” when the rapty location was not picked up by the operator and essentially allow the anky to get away for other anky’s to be eaten instead. The original state of the state machine when given the Anky the state to switch to would just have the Anky spin on the spot, a key press for the Anky to run in a direction was the way I fixed this for the Anky to stop spinning and follow the instructions set by the state machine.(Fig, 5)

```
        // Alerted - up to the student what you do here
        if (anim.GetBool("isAlerted") == true)
        {
            anim.SetBool("isAlerted", false);
            anim.SetBool("isFleeing", true);
```

```
        }
        // Hunting - up to the student what you do here
        if (anim.GetBool("isHunting") == true)
        {
            anim.SetBool("isHunting", false);
            anim.SetBool("isAlerted", true);
        }
        // Fleeing - up to the student what you do here
        if (anim.GetBool("isFleeing") == true)
        {
            set(Ankylocation);
        }
```

(Figure, 4. Test scenario of running states with true and false switches to link factors and functions between states.)

```
        if (Input.GetKeyDown("f"))              //holding f will allow the anky to flee
        {
            anim.Play("Fleeing", -1, 0f);
        }
```

(Figure, 5. The keydown function that allows the anky to flee upon holding.)

Rapty Changes

After assessing the rapty AI behaviour only the follow behaviour linked to another actor in the scene was initialised, this meant an Idle, Hunting and Eating state was to be added to enhance the current mechanic of the Rapty. To do this the location data of the Rapty was needed to utilise the MoveTowards functionality linked with the Anky location to determine the nearest Anky for the Rapty to hunt. Similar to the Anky, a hunger variable was created to meet a condition to move into the Hunting state(Fig, 1). Using that a variable value was needed to enter the state, the same value can be used in an IF statement to start the search parameters and using the Move Towards function to move Rapty to the nearest 'foodSource', Anky. Within the myRapty file, the same box collision parameters are made for the collision between the Anky and Rapty, When entered a debug message of an attack is delivered, with more time adding an attack feature that would lower the health of the Anky by a value for every game tick(Fig, 6).

```
public Transform foodSource;
public float hunger = 0;
hunger = animator.GetFloat("hungerValue");
hunger += 0.8f;
animator.SetFloat("hungerValue", hunger);
if (hunger >= 5)
{
   foodSource = GameObject.FindGameObjectWithTag("Anky").transform;
   animator.transform.position = Vector2.MoveTowards(animator.transform.position,
foodSource.position, speed * Time.deltaTime);
}
```

(Figure, 6. A simple example of the Rapty hunting towards the Anky without the health drop.)

The Rapty animation for attack looked a lot like a spit animation so the attack function is going to need a spit attack for the Rapty. A function called "Spit" is made, this has initialised the and created the projectile using the "GameObeject" class and using the "DealDamageOnCollision" variable lets me set that if it hits the object or in this case the Anky it would detect the hit. By linking the projectile to the vector of the rigidbody of the Anky. Using the float and Vector3 variables to find the Ankys X and Y values for the Rapty to aim the attack was also initialised after the projectile.

The collision data between the Anky and Rapty is debugged in the myRapty file as shown in (Fig, 7). This also shows the TriggerAttack() function that shows the attack animation when an Anky is within the set range parameters, this solves the endless loop of cat and mouse the AI was previously set up to do with an end goal to eat the Anky.

```
private void OnCollisionEnter(Collision colBox)
    {
        if (colBox.gameObject.tag == "deadAnky")
        {
            anim.SetBool("isEating", true);
            anim.SetBool("isHunting", false);
            Destroy(colBox.gameObject);
        }

        else if (toTarget.sqrMagnitude < m_monobehaviour.attackrange)
        {
            m_monobehaviour.TriggerAttack()     #triggers the attack behaviour
            anim.SetBool("isAttacking", true);
            anim.SetBool("isHunting", false);
            Debug.Log("Rapty is attacking an Anky");
        }
    }
```

(Figure, 7. Collision debug and the destroying of a dead Anky.)

A ticker to decrease the hunger is also added when the location of the Rapty is within the foodsource “Anky” shown in (Fig, 8)

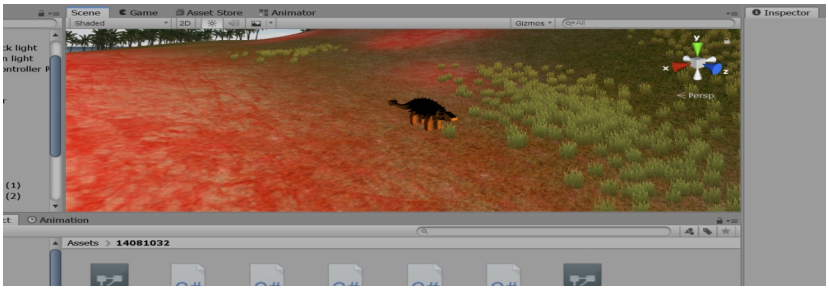
```
            //Decreases the hunger per tick when at foodsource
            if (foodSource.position == Raptylocation)
            {
                hunger -= 5;
            }
```

(Figure, 8. a simple ticker to decrease hunger of the Rapty when at the food source.)

Further developments

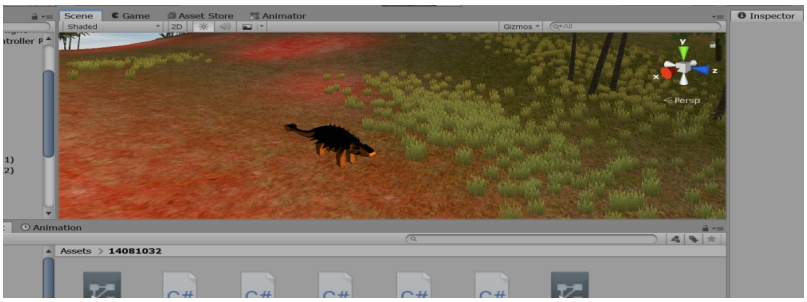
If I had more time I would have been able to fix the errors occurring within the code and with more time I would have created my own state machine as picking up pieces of a purposely broken state machine and to follow the broken code in the same format was difficult. Creating my own state machine would have benefited me further to my own workings and code management. I understand the exercise was to pick up someone else’s work and develop from that.

Appendix



(Anky moves into the box collision)

(Terrain was linked to the box collision, destroying the map terrain from the game)



(Anky successfully destroying a grass object that was created and placed in the grass position on the terrain.)