RUTGERS

Rutgers - Computer Graphics Course - Assignment B5

Unity Behavior Tree

This is a group assignment.

INSTRUCTIONS

In this assignment you must author behaviors **of your choice** using a behavior tree implementation. You will be using a provided library called KADAPT which works in conjunction with TreeSharpPlus.

Follow these instructions:

1. Download the **updated KADAPT** library and set up your group repo.
2. You can run the scene called Scene1.unity to see a character wandering around. There might be some initial build up errors by unity which you can ignore. You can build your own behavior tree on top of this.
3. Look at the notes provided later in this document. You can also check out the original ADAPT documents here:
   https://github.com/ashoulson/ADAPT/blob/master/Unity/Assets/ADAPT%20Core/Tutorials/Tutorial1-4.pdf
   Note that the documents in the above link are out of date, but the concepts and architecture of the library are still the same, some implementations have been updated to support Mecanim animation system.
4. Think of a story you want to author. This can be anything you want, but it must satisfy these criteria:
   a. At least three interacting characters must be involved.
   b. The environment must be complex (you can use your environment from previous unity assignments).
   c. The story must have a beginning, middle, and an end (immersive characteristic).
   d. Some behaviors within the story must change on each run (non-linear characteristic).
   e. The behavior must be scalable to many agents, i.e., it must incorporate parameterizable behavior trees (scalability characteristic).
   f. There must be more than one ending (interactivity characteristic).

5. **[4 points]** Implement at least 2 new affordances (atomic interactions between characters and/or objects) of your own that are NOT provided in the KADAPT library. Example affordances include picking up / dropping an object, opening / closing a door, press a button, a specific movement pattern, a dialogue pop-up, etc. These affordances may or may not use Inverse Kinematics. Use them in your story.

   After you decide on your behavior, design the behavior tree implementation for that behavior. Include your behavior tree drawing in your report, and justify the use of each control node you have used.
6. **[4 points]** Implement your behavior tree using KADAPT library. The result is a game which uses your behavior tree to create your desired behavior.
7. **[4 points]** Use Inverse Kinematics to create 2 additional affordances for use in your behavior (e.g. throw/catch a ball, two characters shaking hands). You must create some new nodes which can handle object movement, and then you can use KADAPT and its methods to make any one of your characters interact with the object using Inverse Kinematics.
8. **[2 point]** Create a specific control node of your own, and assign appropriate behavior with it. Use the structure of KADAPT node implementation to create your own node. Describe why having this node can be advantageous, and use it in your behavior.
9. **[6 point]** Design a complete **Interactive Behavior Tree** for your story, same as provided in narrative slides (see Interactive Behavior Tree implementation). Include the drawing in your report. Note that the design does not have to be exactly the same, but must have same architecture and satisfy same objectives.

   Attach human player controls to one character (assume a character as the human interactor), and modify your interactive tree accordingly.

   The player must now be able to interact with your story, causing changes in both flow and ending of your game. Your story must have multiple endings.
10. We expect a clear explanation of your 4 new affordances (2 general, 2 IK) in your report, as well as drawing of your behavior tree with sufficient justifications on the design. Points will be deducted if any part is not clear.

KADAPT Architecture

KADAPT is simply a modified version of ADAPT which uses Mecanim animation system. Here we will go through some basic information regarding how KADAPT works.

Animation:

There are three main objects that handle the animation process of a character: BodyMecanim, CharacterMecanim, and BehaviorMecanim. You can find these under the the path "Core\Scripts\Character".

1. **BodyMecanim:** this is where the main motor skills and animation core is located. Low level implementations of how a character moves can be found here (e.g. navigation, steering, IK, etc.). Look at BodyMecanim.cs for more detailed information.
2. **CharacterMecanim:** This is basically a wrapper around BodyMecanim functionalities. Here higher level actions are implemented which use several methods provided in BodyMecanim together to create an actual behavior. Combinatorial actions can be created here for any character. Look at CharacterMecanim.cs for more detailed information.
3. **Behavior Mecanim:** This is where subtrees of high level behaviors are implemented, using the provided methods in CharacterMecanim. Here you can find high level action nodes that can be used as leaf nodes in your tree (e.g. Node_GoTo, ST_PlayGesture, etc.). Look at BehaviorMecanim.cs to see all possible leaf nodes that are provided to you.

Note that in order for these classes and methods to work, you must have a special character animation and mecanim state machine set up. In other words, the animation implementation supposes a specific design of an animator controller. So use the provided prefab character in the folder "KADAPT\KadaptPrefab" for your implementation. You can also design your own character by look at the animator and the controller for these provided prefabs and imitating them.

Behavior Tree:

Core functionality of behavior trees are implemented In TreeSharpPlus, which can be found under "**Assets\Core\Scripts\Behavior\TreeSharpPlus**". In TreeSharpPlus, behavior trees are treated as their own little processes. A behavior tree is automatically ticked on a regular basis by the Behavior Manager. During each tick, the behavior tree evaluates whatever node is currently active, and is told whether that node has succeeded, failed, or is still running and should be ticked again next round. The success or failure of a node, once it's terminated, allows the behavior tree to pick the next node to execute depending on the structure of its control nodes (like sequences and selectors). Nearly every function having to do with running a behavior tree in TreeSharpPlus returns a RunStatus, which is an enumerated type containing

“Success”, “Failure”, or “Running”. This is how nodes in the tree hierarchy communicate with one another. Look at Node.cs and NodeGroup.cs to get an idea of the backbone implementation of each node. There are two main components you need to look at:

1. **Control Nodes:** These are mostly derived from GroupNode class, and can be used to control the flow of your behavior. Examples are Selector.cs, Sequence.cs, DecoratorLoop, etc.
2. **Leaf Nodes:** These are mostly derived from Node class, and can be used as effector nodes, containing functions or assertions. Examples are LeafInvoke.cs, LeafTrace.cs, LeafWait.cs, LeafAssert.cs, etc.

Note that each can be Ticked or Terminated, these are the essential functionalities of each node.

Procedure:

So how does it all work? There is no simple answer to this question, as this implementation is somewhat complex. However, we can simplify things as follows (look at the code and you can make sense of each step, look at the scripts at “**Assets/Core/Scripts/Behavior**”):

1. **Behavior Updater:** This is the object responsible for ticking all the trees, i.e. this is the overall clock generator. This object runs the Update method of the Behavior Manager.
2. **Behavior Manager:** This is the object that holds the trees in your scene, and runs their BehaviorUpdate method whenever instructed by the Behavior Updater (basically this passes the ticks coming from updater to all the receivers). Note that there can never be more than one instance of this Behavior Manager in a scene.
3. **Behavior Agent:** This is one of the objects that can hold your tree (i.e. your tree root). This object handles high level operations on the tree it owns. This is basically just a wrapper around your tree. You must register this object to the Behavior Manager, so it can receive and pass on the ticks to the tree. When registered, at the call of its BehaviorUpdate method it will tick the tree (or terminate it!).
4. **Root Node:** This is what you create, using control nodes (sequence, selector, etc.) and leaf nodes (LeafInvoke, LeafAssert, etc.) and pass it to the Behavior Agent. Each time the root ticks, it will tick its children, if any, and the ticks continue down the tree until reaching a leaf to execute. This process is implemented using Closures and Enumerators in C#.

This is a pretty general overview just so you can start your project. If you need more insights into how things work, first read the following documents, and then start looking at the implementation of each part discussed above carefully.

Must Read Materials:

1. Closures in C# overview:
   https://richnewman.wordpress.com/2011/08/06/closures-in-c/
2. IEnumerator<T> overview:
   https://msdn.microsoft.com/en-us/library/78dfe2yb(v=vs.110).aspx
   http://stackoverflow.com/questions/558304/can-anyone-explain-ienumerable-and-ienumerator-to-me
3. Yield in foreach:
   https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx

Walkthrough

In this part we will go through a simple tutorial for how to use KADAPT. you can find the tutorial file in KADAPT project under "_Scenes\scene1.unity". So here is how it works:

1. Create an environment with some obstacles.
2. Bring in the prefab character named "Daniel". Look at how Daniel is designed, look at all the scripts attached to it which take care of different behaviors for Daniel.
3. Look at the animator and the state machine attached to Daniel. The layers of the animator controller are used to blend different animations together (e.g. running and IK).
4. Bake a navigation mesh so Daniel's NavMeshAgent can use it to navigate.
5. Create an empty object in your scene ("Updater"), and attach a Behavior Updater script to it (from Core\Scripts\Behavior). Set up the Update Time variable to your liking (this is how fast it ticks your trees).
6. Now you must create your behavior tree and put it in your scene. Here, we want a wandering behavior for a character, that means we want the character to wander between three user defined points. So create a script as follows and attach it to something in your scene (preferably your updater, or director). Note that it will work the same no matter where you attach it. Look at "Assets\MyBehaviorTree.cs".
7. Assign the Wander points (create three empty object), and also your character to the script.

8. Run the game, see how Daniel wanders between these points. If you move the points, his wandering will adapt, that means you can interact with him! This is how closures help us create a Parametric Behavior Tree.
9. Read the fourth tutorial in the original tutorial document for ADAPT, mentioned at the top, to see how the closures (implemented within a type called Val) are helping us create the interactive process.
10. You can author your behavior same way, so go ahead and get creative. Remember you can handle objects same way as we are handling a character, just implement the required Body, Character, and Behavior Nodes for the object.
11. Also take a look at "_Scenes\scene2.unity" and "Assets\MyBehaviorTree2.cs" for a more complicated example. See how initially Daniel is idle: try moving the "Police1" agent in the environment passed 10 on z axis and see what happens. You can move police1 back and forth as the simulation runs.

```
using UnityEngine;
using System.Collections;
using TreeSharpPlus;

public class MyBehaviorTree : MonoBehaviour
{
    public Transform wander1;
    public Transform wander2;
    public Transform wander3;
    public GameObject participant;

    private BehaviorAgent behaviorAgent;
    // Use this for initialization
    void Start ()
    {
        behaviorAgent = new BehaviorAgent (this.BuildTreeRoot ());
        BehaviorManager.Instance.Register (behaviorAgent);
        behaviorAgent.StartBehavior ();
    }

    // Update is called once per frame
    void Update ()
    {

    }

    protected Node ST_ApproachAndWait(Transform target)
    {
        Val<Vector3> position = Val.V (() => target.position);
        return new Sequence( participant.GetComponent<BehaviorMecanim>().Node_GoTo(position), new LeafWait(1000));
    }

    protected Node BuildTreeRoot()
    {
        return
            new DecoratorLoop(
                new SequenceShuffle(
                    this.ST_ApproachAndWait(this.wander1),
                    this.ST_ApproachAndWait(this.wander2),
                    this.ST_ApproachAndWait(this.wander3)));
    }
}
```

The above code is a simple wandering script, as can be found in MyBehaviorTree.cs code.

SUBMISSION

Submit the following in Sakai for grading:

1. Your Unity project in a zip file which contains the **Assets/Scripts** folder and includes all your C# scripts (Not the whole KADAPT library, just any script you implemented or changed).
2. Offline build of your game (Win x86_64).
3. Brief documentation about your project.
    a. Affordance descriptions and implementation details
    b. Behavior Tree description and drawing of the tree
    c. Other relevant information
4. Link to a video demo.
5. A log of your git commit history.

General Grading Criteria

1. Behavior Complexity and Nonlinearity
2. Behavior Tree Design
3. Behavior Tree Implementations
4. Creativity

NOTE: Extra credit will be given at the discretion of the instructor.

--

Good Luck Everyone!