REALMEMORY QUICK(ISH) START GUIDE 1.1

UNITY SETUP

It's easy to set up RealMemory! There are two ways, depending on how you downloaded the engine—through the asset store or as a unitypackage file.

If you bought RealMemory through the Asset Store, simply follow the import directions there. The dlls and resources will be imported into their appropriate folders automatically.

If you have a unitypackage file, simply import the unitypackage into your project. This can be done in several ways; for instance, by going to Assets->Import Package and choosing Custom Package, then selecting the RealMemory unitypackage and clicking Open. When the Importing Package window pops up, click Import.

Regardless of how you've imported RealMemory, you may need to click on the menu bar in Unity to get the Tools/RealMemory menu option to show up for the first time. Also, to successfully build a project using RealMemory, you'll need to change the PlayerSettings in Unity (Edit->Project Settings->Player, or click Player Settings from the Build Settings dialogue box) setting for API Compatibility Level (at the bottom, under Optimization) to .NET 2.0 (NOT the default ".NET 2.0 Subset"). The subset lacks the functionality necessary to read some of the dlls used after compiling.

That's it! You're ready to create some memories!

FROM THE SIMPLE TO THE COMPLEX

RealMemory has three levels of complexity for use in a variety of gaming situations—stories only, using long-term memory and stories, or (the most realistic) using both the short-term and long-term memory systems and stories. (Actually, you can also forego the use of stories in the latter two situations, if you like.) We'll start with stories only as a way of introducing some elements of the RealMemory system.

THE SIMPLE: STORIES ONLY

CREATE A CHARACTER AND STORIES IN THE EDITOR

Go to the Tools menu, choose RealMemory, and then choose the Stories submenu. The Add or Edit a Story window opens. In the Choose Character drop-down, you'll see the two pre-packaged characters, Bard and Healer. There is also a text box in which you can add a new character. Let's add Buttons. Type "Buttons" into the box and click the Find Character's Stories button.

As you'd expect, this character has no stories—yet. Add one by clicking Add New Story, then typing "The ocean is great!" into the text box that appears, then clicking Create Story. Leave 50 as the story strength (the value could be between 0 (weakly remembered) and 100 (strongly remembered)) and click Done (save) in the Add New Story Text box (NOT Done (save changes) in the main window—that will save and close the Add or Edit a Story window).

Now Buttons knows a story, but he doesn't have any context for it (that is, he can't recognize how the story might relate to anything). To the right of the story itself is an empty drop-down with an Add Concept button. Clicking this, we can add "ocean" to the story's concept list. Do this by typing it into the text box that appears in the Add Concept to … subwindow, then clicking Add new!

The Types box lets you add a tag to help group stories into categories; for instance, one tag that is already available is "Opinion," meaning that this story represents the character's opinion about something, which seems

appropriate for our example. Click Add Type, then select Opinion from the drop-down list in the subwindow and click Add Existing.

Finally, let's give this story an emotional tie, meaning that the story might incite certain feelings in the character or might be remembered if certain feelings are aroused. Click Add Emotion, and then select Happiness from the drop-down menu in the subwindow and click Add Existing.

Now click Done (save changes) in the Add or Edit Story window, and now Buttons is of the opinion that the ocean is great!

Checking Stories During Play

Let's create a simple script to access Button's memories in-game.

1. Create a new script (for example, a C# script called "NPCInterface").
2. In the script, add "using RealMemory_Lib;" to the other "using …" statements at the top
3. Add "public AIMemSensory myMem = new AIMemSensory();" to your variables
4. Add "string myName = [your character's name]" to your variables. For example, this could be string myName = "Buttons";
5. In the Start method, initialize the engine by adding "myMem.Init (this.tag,myName);" Note that this is also where you'd first pass certain personality values if you're using a personality engine (such as Extreme AI). As it is, Buttons will be given default personality values
6. Save the script
7. Add the script to the GameObject representing your NPC. (Make sure the GameObject has a tag so the Init call above works!)

Because we'll be using lists later in this Guide, you should also add "using System.Collections.Generic;" to the "using …" statements at the top of your script.

Let's Ask About the Ocean

So now we encounter Buttons walking down the street and, since we're wondering what to do with our day, we ask him whether he has any opinions about the ocean. For simplicity, let's just add this to the NPCInterface script (although it would more likely come from elsewhere in an actual game). We'll use:

```
myMem.StoryRetOnConcept(listOfConcepts, ref oceanStories);
myMem.StoryRetOnType(listOfTypes, ref oceanOpinions);
```

And compare them to see whether Buttons has any opinions about the ocean. Note that the variables being passed in these methods are string arrays; in this example, listOfConcepts contains only one string ("ocean"), and listOfTypes contains only one string ("opinion"). The other two arrays are empty; they will be populated by the method itself. In full, the NPCInterface script would look like:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using RealMemory_Lib;

public class NPCInterface : MonoBehaviour {

    public AIMemSensory myMem = new AIMemSensory();
    string myName = "Buttons";

    // Use this for initialization
    void Start () {
```

```
        myMem.Init (this.tag,myName);

        string[] oceanStories = new string[0];
        string[] oceanOpinions = new string[0];
        string[] listOfConcepts = new string[] {"ocean"};
        string[] listOfTypes = new string[] {"opinion"};

        myMem.StoryRetOnConcept(listOfConcepts,ref oceanStories);
        myMem.StoryRetOnType(listOfTypes, ref oceanOpinions);

        //find only the ocean stories that are opinions and print in debug.log
        foreach(string oceanStory in oceanStories)
        {
            foreach(string oceanOpinion in oceanOpinions)
            {
                if(oceanStory==oceanOpinion)
                {
                    Debug.Log (oceanStory);
                }
            }
        }
    }

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

    }
}
```

Create an empty GameObject and attach the NPCInterface script to it. When you run the game, the log should reveal that Buttons has an opinion about the ocean: It is great!

More Complexity: Using the LTM

A more realistic way of looking at character memories is through using the LTM (and stories); in fact, even when you're using stories, RealMemory is using the LTM in the background (that's where the story concepts are stored). In RealMemory, each memory is a concept—a small bit of information, such as the color blue, or the clothing item shirt, or the character's best friend Jeffrey. These concepts are related to other concepts in various ways. Some concepts interconnect in a hierarchical fashion: Jeffrey has eyes, hair, skin, etc. Eyes in general have color and shape. Jeffrey's eyes may be green and round.

Using the Editor to Edit the LTM

To continue with Buttons, let's add a memory by going to Tools->RealMemory->Memories. This opens the Memories window. Choose Buttons from the Choose Character drop-down, then click Find/Create Memories.

You'll see that, because we created a story about the ocean (and added the story concept "ocean"), Buttons already has "ocean" in his LTM (showing up in the top section, Memory List). But what if we want him to think of the ocean as "blue"?

Let's add "blue" to the Memory list. Click Add/Chg Mem, then in the subwindow that appears, type "blue" in the New Memory text box and click Continue. We are given the option to give this memory a strength and an emotional charge. For now, though, we'll leave these as-is and click Add Concept.

Buttons now knows about blue and ocean. We could just connect these directly, making blue a related concept of ocean, but that doesn't quite make sense. What's really going on is Buttons knows that the ocean has a color, and that color is blue.

This is where we use the Ur list section. What we'll do is add the overarching ("Ur") concept "color," then add blue to the list of colors, then go from there.

To do this, go to the lower section (Buttons Ur list) and click Add Ur. Type "color" in the subwindow that appears, then click Create Ur. We now have the overarching category "color"! We now want to add members (colors) to this category (skipping the subconcepts); go to the next row, and click Add Member. Click "blue" in the drop-down list, then click Add Member.

Now that Buttons knows of colors, and knows that one color is the color blue, we can add this information to the concept of "ocean." Go back to the Memory list, choose "ocean" from the drop-down, and click Add Conc (meaning add concept) below the second box. In the subwindow, choose "color" from the drop-down and click Add Concept. Another drop-down appears allowing you to add properties to this concept. Choose "blue" from the list, then click Done (save)! You have now added a complete simple memory: Buttons knows that the ocean has a color, and that color is blue. He also knows of the idea of colors, and that one of these colors is blue, an example of which is the color of the ocean. You'll notice that he also still remembers the story about the ocean, believing that the ocean is great.

Click Done (save changes) at the bottom of the Memories window to save this work.

Note that, because this process can be time-consuming, we have created memory packs that instantly teach the characters all they need to know about certain general subjects (like colors). You can buy these memory packs from our site or from the Asset Store (or contact us to ask if we can create one for you). One Memory Pack is included with RealMemory: MemPackDemo. To install this memory pack, go to Tools->RealMemory->Install MemPack. This will bring up a window allowing you to choose which memory pack to use and which character to give the memories to (including a new character or all existing characters). Choose the Demo pack, choose Buttons, and click the Install Memories button. Click Done to exit the window. If you look at Buttons' memories, you'll see that he now knows a lot more than he did before!

You can also add items directly to the LTM during gameplay, but that's not covered in this quick start guide; see the User Manual for how to do this.

Asking Buttons about the Ocean, Again

This time we want to ask Buttons if he knows what color the ocean is (if he thinks the ocean is brown or black, we might think twice about going into it). Add the following to our previous script (at the end of the Start method):

```
//using the LTM for recall
string[] oceanProperties = new string[0];
float[] oceanPropStrengths = new float[0];

string whatColor = myMem.MemRetOneKnown("color","ocean", ref oceanProperties, ref
        oceanPropStrengths);

Debug.Log ("whatColor: " + whatColor);

foreach(string oceanProperty in oceanProperties)
{
        Debug.Log ("The ocean is " + oceanProperty + ".");
}
```

Here we are using MemRetOneKnown to return whether Buttons knows “color” about “ocean,” and providing arrays for the color properties (in this case, just “blue”) and strengths (which we don’t need in this case, but retrieving them is part of the method). We’re then printing this information in the log. Note that the method returns "I know about that!" if the character knows about color as it relates to ocean, in addition to returning the actual properties by ref.

And the Most Complex: Using the STM and LTM

Sending Items to the STM

The most realistic way of using RealMemory is to use the short- and long-term memories in addition to stories. You act as the sensory input for your NPC; RealMemory does the rest. In some ways this could be easier than using the LTM directly, if you can provide some kind of automated sensory interface.

Characters using RealMemory have a short-term memory system (STM) similar to that of a human being. They can remember 5-9 distinct items at a time, and these will either fade from the STM over time or be strong enough to be encoded into long-term memory (LTM).

Let’s say Buttons is spending a day at the beach. He sees and smells the ocean, and kind of notices a sand castle a child has built, but is primarily focused on the ocean itself. (This is all entered during gameplay, not in the editor.) This is more complex than stories; we need to input the following:

- The memory itself
- Type of input (visual, auditory, olfactory [smell], or combinations of these)
- Subconcepts of this memory (for instance, “ocean” might have wetness, color, and more)
- Any context that might be associated with this memory (the ocean might remind the character of childhood, for instance)
- Amount of attention the NPC is paying to the memory
- Surrounding environment
- Motivation to remember this memory
- Any source of the memory (if, say, the NPC is being told about the ocean rather than experiencing it directly)
- Emotional charge (the amount of emotional charge, which makes the charged object more memorable and surrounding objects less memorable)
- Emotional association (e.g., the sight of the ocean makes the NPC feel incredibly happy)

The most complicated part of this is the subconcepts; this is a list of objects that include the subconcept itself, its strength of association with the main memory, and any further properties of the subconcept. For instance, the ocean may have:

Subconcept: color
Strength: 100
Property: blue

Subconcept: wetness
Strength: 100
Property: wet

You can create the list of subconcept objects as follows (add this code to the bottom of your Start method):

```
List<MemConceptGroup> relatedConcepts = new List<MemConceptGroup>();

relatedConcepts.Insert (0, new MemConceptGroup{memConcept="color",memConStrength=100});
```

```
relatedConcepts[0].memProperties.Add (new Properties{propName="blue",propUr="color"});

relatedConcepts.Insert (0, new MemConceptGroup{memConcept="wetness",memConStrength=100});
relatedConcepts[0].memProperties.Add (new Properties{propName="wet",propUr="wetness"});
```

Another list you need to send is the anyContext list. It's much simpler than the first one, consisting only of a string representing the context of the memory and a float representing the context strength (0-100). The context strength will default to 50 if you don't include it. You can create this list as follows:

```
List<ContextItems> thisContext = new List<ContextItems>();
thisContext.Add (new ContextItems{contextName="Fun Beach",contextStrength=80});
```

You can then use the remaining elements to indicate how strong this memory should be in the STM, which affects whether it gets to the LTM. In this case, let's make Buttons' attention level strong, most other things average, and the emotional charge slightly high (due to the excitement of being at the ocean):

```
myMem.SensoryMemory ("ocean", "threeAVS", relatedConcepts, thisContext, 1, 0, 0, "", 1);
```

See the User Manual for a more complete explanation and the ranges for the variables in this method.

For the sand castle, we do the same things. Let's say the subconcept is color (it could of course be more, but for our example just use color), which is golden:

```
List<MemConceptGroup> relatedCastleConcepts = new List<MemConceptGroup>();

relatedCastleConcepts.Insert (0, new MemConceptGroup{memConcept="color",memConStrength=100});
relatedCastleConcepts [0].memProperties.Add (new Properties{propName="golden",propUr="color"});
```

And the context is the same:

```
List<ContextItems> thisCastleContext = new List<ContextItems>();
thisCastleContext.Add (new ContextItems{contextName="Fun Beach",contextStrength=80});
```

But in this case less attention is being paid, and the emotional charge should be negative; also, there is only a visual element:

```
myMem.SensoryMemory ("sand castle", "visual", relatedCastleConcepts, thisCastleContext, -1, 0, 0, "", -1);
```

TESTING THE STM

In order to properly test the STM, we need to wait for Buttons' STM to empty (meaning there are no more potential memories waiting to either get into the LTM or fade away). To do this, we can use MemTimeCheck, which returns false if there are no items in the STM. We'll add the following to the variable declaration section of the script:

```
bool continueChecking = true;
float timeBetweenMemPings = 0.0f;
float timeToCheckMem = 2.0f; //set to check STM every x seconds
float totalTime = 0.0f; //keeping track of time passed
bool alreadyPrinted = false;
```

And add the following to the Update method (which checks that the STM is empty, and then prints to the log using the same recall method as in the LTM section):

```
if(continueChecking)
{
    timeBetweenMemPings += Time.deltaTime;
    totalTime += Time.deltaTime;

    if(timeBetweenMemPings>=timeToCheckMem)
    {
        continueChecking = myMem.MemTimeCheck (timeBetweenMemPings);
    }
} else {
    if(!alreadyPrinted)
    {
        Debug.Log ("Buttons' STM is empty");

        string[] oceanProperties = new string[0];
        float[] oceanPropStrengths = new float[0];

            string whatColor = myMem.MemRetOneKnown("color","ocean", ref oceanProperties, ref
                oceanPropStrengths);

        Debug.Log ("whatColor: " + whatColor);

        foreach(string oceanProperty in oceanProperties)
        {
            Debug.Log ("The ocean is " + oceanProperty + ".");
        }

        string[] oceanWetProperties = new string[0];
        float[] oceanWetPropStrengths = new float[0];

            string howWet = myMem.MemRetOneKnown("wetness","ocean", ref oceanWetProperties,
                ref oceanWetPropStrengths);

        Debug.Log ("howWet: " + howWet);

        foreach(string oceanWetProperty in oceanWetProperties)
        {
            Debug.Log ("The ocean is " + oceanWetProperty + ".");
        }

        string[] castleProperties = new string[0];
        float[] castlePropStrengths = new float[0];

            string castleColor = myMem.MemRetOneKnown("color","sand castle", ref castleProperties,
                ref castlePropStrengths);

        Debug.Log ("castleColor: " + castleColor);

        foreach(string castleProperty in castleProperties)
        {
            Debug.Log ("The sand castle is " + castleProperty + ".");
        }

        alreadyPrinted = true;
    }
}
```

For convenience, the entire script is provided at the end of this guide.

Note that the STM is set to empty much more quickly than it would in the real world; this can be changed using the SecsBetChecksOverride method. For instance, you could add

```
SecsBetChecksOverride(0.5f);
```

This would make the STM empty much, much faster (and is useful in many situations, such as when the NPCs are observing something before the players arrive). Don't worry, though—their entire memory process is sped up, so they won't lose anything they'd remember at a slower speed.

Given these parameters, Buttons will likely remember the ocean, but not the sand castle. However, you can play around with the parameters and change what he remembers and what he doesn't.

And that's it! You've created a character, given him memories using a variety of methods, and asked him about his memories. There is of course much, much more you can do with RealMemory; see the User Manual and (soon) our YouTube channel for further examples. And feel free to contact us with any questions or requests at support@quantumtigergames.com.

COMPLETE NPCINTERFACE SCRIPT

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using RealMemory_Lib;

public class NPCInterface : MonoBehaviour {

    public AIMemSensory myMem = new AIMemSensory();
    string myName = "Buttons";

    bool continueChecking = true;
    float timeBetweenMemPings = 0.0f;
    float timeToCheckMem = 2.0f; //set to check STM every x seconds
    float totalTime = 0.0f; //keeping track of time passed
    bool alreadyPrinted = false;

    // Use this for initialization
    void Start () {
        myMem.Init (this.tag,myName);

        string[] oceanStories = new string[0];
        string[] oceanOpinions = new string[0];
        string[] listOfConcepts = new string[] {"ocean"};
        string[] listOfTypes = new string[] {"opinion"};

        myMem.StoryRetOnConcept(listOfConcepts,ref oceanStories);
        myMem.StoryRetOnType(listOfTypes, ref oceanOpinions);

        //find only the ocean stories that are opinions and print in debug.log
        foreach(string oceanStory in oceanStories)
        {
            foreach(string oceanOpinion in oceanOpinions)
```

```
        {
            if(oceanStory==oceanOpinion)
            {
                Debug.Log (oceanStory);
            }
        }
    }


    //using the LTM for recall
    string[] oceanProperties = new string[0];
    float[] oceanPropStrengths = new float[0];

    string whatColor = myMem.MemRetOneKnown("color","ocean", ref oceanProperties,
        ref oceanPropStrengths);

    Debug.Log ("whatColor: " + whatColor);

    foreach(string oceanProperty in oceanProperties)
    {
        Debug.Log ("The ocean is " + oceanProperty + ".");
    }


    //using the STM to provide sensory information
    List<MemConceptGroup> relatedConcepts = new List<MemConceptGroup>();

    relatedConcepts.Insert (0, new MemConceptGroup{memConcept="color",memConStrength=100});
    relatedConcepts[0].memProperties.Add (new Properties{propName="blue",propUr="color"});

    relatedConcepts.Insert (0, new MemConceptGroup{memConcept="wetness",
        memConStrength=100});
    relatedConcepts[0].memProperties.Add (new Properties{propName="wet",propUr="wetness"});

    List<ContextItems> thisContext = new List<ContextItems>();
    thisContext.Add (new ContextItems{contextName="Fun Beach",contextStrength=80});

    myMem.SensoryMemory ("ocean", "threeAVS", relatedConcepts, thisContext, 1, 0, 0, "", 1);

    List<MemConceptGroup> relatedCastleConcepts = new List<MemConceptGroup>();

    relatedCastleConcepts.Insert (0, new MemConceptGroup{memConcept="color",
        memConStrength=100});
    relatedCastleConcepts [0].memProperties.Add (new Properties{propName="golden",
        propUr="color"});

    List<ContextItems> thisCastleContext = new List<ContextItems>();
    thisCastleContext.Add (new ContextItems{contextName="Fun Beach",contextStrength=80});

    myMem.SensoryMemory ("sand castle", "visual", relatedCastleConcepts, thisCastleContext, -1, 0, 0,
         "", -1);
}

void Update () {
    if(continueChecking)
    {
```

```
            timeBetweenMemPings += Time.deltaTime;
            totalTime += Time.deltaTime;

            if(timeBetweenMemPings>=timeToCheckMem)
            {
                continueChecking = myMem.MemTimeCheck (timeBetweenMemPings);
            }
        } else {
            if(!alreadyPrinted)
            {
                Debug.Log ("Buttons' STM is empty");

                string[] oceanProperties = new string[0];
                float[] oceanPropStrengths = new float[0];

                string whatColor = myMem.MemRetOneKnown("color","ocean", ref oceanProperties,
                    ref oceanPropStrengths);

                Debug.Log ("whatColor: " + whatColor);

                foreach(string oceanProperty in oceanProperties)
                {
                    Debug.Log ("The ocean is " + oceanProperty + ".");
                }

                string[] oceanWetProperties = new string[0];
                float[] oceanWetPropStrengths = new float[0];

                string howWet = myMem.MemRetOneKnown("wetness","ocean", ref oceanWetProperties,
                    ref oceanWetPropStrengths);

                Debug.Log ("howWet: " + howWet);

                foreach(string oceanWetProperty in oceanWetProperties)
                {
                    Debug.Log ("The ocean is " + oceanWetProperty + ".");
                }

                string[] castleProperties = new string[0];
                float[] castlePropStrengths = new float[0];

                string castleColor = myMem.MemRetOneKnown("color","sand castle", ref castleProperties,
                    ref castlePropStrengths);

                Debug.Log ("castleColor: " + castleColor);

                foreach(string castleProperty in castleProperties)
                {
                    Debug.Log ("The sand castle is " + castleProperty + ".");
                }

                alreadyPrinted = true;
            }
        }
    }
}
```