Extreme AI: RealMemory

User Manual v1.1, June 2016

Welcome to RealMemory, our second in a line of projects designed to give independent realism to game characters. Like the Extreme AI Personality Engine, RealMemory gives your NPCs a chance to be a little more human, react in human-like ways, and provide your players with a more immersive game experience.

With RealMemory, your NPCs have human-like memory systems, complete with short-term and long-term memories. They can remember, forget, and distort memories, suddenly remember a "forgotten" bit of information when given contextual clues, and react to emotionally charged memories. They can also store short "stories" as simple memories to supplement the STM and LTM systems.

Think of an NPC at the scene of a crime, having just witnessed a murder. With RealMemory, the NPC will be able to remember some of the events more than others, some items more clearly, some with great emotion, not remember some clues at all, and misremember others. And different witnesses will remember events and things differently. "No, really, I'm SURE he had orange hair and a moustache!" "He couldn't have. I remember his face being as smooth as a fresh sheet of ice." And so forth.

RealMemory bases some of its algorithms on NPC personality traits, so ideally you should have a personality engine like Extreme AI to pass on those values. (Note that you can get the personality engine at a discount if you have RealMemory!) If you do not pass any personality values to RealMemory, it will use default values and, while still working, won't be customized to each character in quite the same way.

Some specific things you can do with RealMemory:

- Give NPCs short-term memories that can store a limited amount of information that decays or is strengthened and moved into the long-term memory in ways similar to human STM
- Remember in LTM any number of memory concepts, tied together into related concepts and properties thereof; for example, "Jeffrey" is a memory concept that is part of the overarching concept "person" and has "eyes" that are "green" (or are they "blue"?)
- Give memories emotional ties
- Distort memories
- "Forget" (lose access to) memories
- Use contextual clues to "jog" an NPC's memory
- Associate stories with memory concepts

You have control

As with our other products, we try to give you as much control over the actual processes occurring as you might want, from allowing the NPC's mind to churn on its own to using only the bits you like. If you don't want to use the short-term memory functions, that's fine—you can inject memories directly into the LTM. If you'd like some memories to be stored permanently, with no distortion or obfuscation, that's fine, too. And at its simplest, you could choose to use only the Stories a character knows and not individual memories. But if you want the NPC's mind to run on its own, from STM to LTM to forgetfulness and distortion to everything else, you can have it run more-or-less automatically. You do need to provide the information to the STM (you have to be the NPC's senses, more or less), and you tell the NPC how often to perform "maintenance checks" on her memory (as she cannot tell whether days or years have passed in the game world since her last check, unless your game is continuously in real time), but otherwise the processes can work on their own.

Where did all this come from?
As with ExtremeAI, I've spent years researching this stuff, sometimes in an academic environment, sometimes on my own.

In a nutshell, I've looked at modern theories of how memory works and attempted to translate these into code that can be used by game developers. While this necessitated some "fudging" in order to get things to work for games, and some guesswork (as we don't know for certain how some parts of memory actually work), I believe I've achieved something that at the very least looks and acts like a human memory system, even if the inner workings are more computer than chemical.

As usual, thank you very much for taking a look at RealMemory, and please let us know what you're doing with it! We're happy to advertise games (and other things) that use our products. And feel free to ask any questions you may have.

Jeffrey Georgeson
Quantum Tiger Games

CONTENTS

Installation ... 1

- System requirements ... 1
- Initial Setup for Unity ... 1

Using RealMemory: In General ... 2

- Overview ... 2
- Initializing RealMemory ... 2
- Memories Based on Personality? ... 3
- How Do I Change Personality Values In-Game? ... 4

In-Game Use of RealMemory: From the Simple to the Complex ... 5

- Using Only Stories ... 5
  - Adding a Story to Memory ... 5
- Weakening Memories through Remembering Others? ... 6
  - My Head Is Filled with Stories, but How Do I Get Them Out? ... 6
  - I know the story ... 7
  - Other Story-Related Methods ... 7
  - That reminds me … ... 7
  - The Years Go By: Memory Maintenance ... 8
  - Saving Stories ... 9
- Using the LTM ... 9
  - An example ... 10
  - Another Example ... 11
  - Can't this @$#% method call be made any simpler? ... 12
  - Adding Other Elements to a Memory after It's Formed in the LTM ... 12
  - Adding Context ... 12
  - Conflicting information ... 13
  - Asking a Character Questions: Retrieving Memories from the LTM ... 13
  - Memory Maintenance (and distortion and obfuscation) ... 14
  - Other LTM-related methods ... 15
  - Saving Memories ... 15
- Real RealMemory: Using the Short-Term Memory (and LTM and Stories) ... 15
  - Sensory Memory ... 15
  - An example ... 17
  - Can't this @$#% method call be made any simpler? ... 18

What happens inside the STM? 18
It's Taking So Long to Empty the STM … 19
Creating Memories with the Editor 20
The Concept of Concepts 20
Using the Editor (Creating Characters Prior to Gameplay) (Advanced Editing) 20
A note on setting up characters 20
Deleting a Character 20
Adding Memories the Easy Way (Memory Packs) 21
Adding Individual Memories: The Memories Window 21
Telling Tales: The Stories Window 26
Create Memory from Overarching Concept: The Create from Ur Window 27
Duplication: The Copy Memories Window 28
Register RealMemory! 29
Support and Credits 30
Support 30
Credits 30

INSTALLATION

SYSTEM REQUIREMENTS

The Unity-specific version of RealMemory works in Unity 5 and above. Otherwise, the disk space, memory, etc. requirements are minimal.

INITIAL SETUP FOR UNITY

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!

USING REALMEMORY: IN GENERAL

OVERVIEW

RealMemory gives your NPCs human-like memory systems they can use to recall concepts and related information. They can also forget or distort information, have emotional reactions to it, use context to try to jog their memories, and associate stories with certain memories.

NPCs using all aspects of RealMemory have short-term memories (STMs) and long-term memories (LTMs). Information coming into the STM has a limited lifespan, and can either expire after a certain time or be forced out by other incoming information. Like humans, NPCs with RealMemory can hold between 5 and 9 pieces of information in STM at any one time; this is in part determined by certain personality factors (represented by an NPC's instance of the Extreme AI personality engine). If a potential memory is strengthened in some way (e.g., through repetition or by being similar to an existing memory in the LTM[1]) or is strong in the first place (e.g., through a strong emotional tie), it will be moved to the LTM.

Memories in the LTM are generally there forever (as in some theories of human LTM), but may be distorted or become irretrievable (obfuscated) in some way over time, due to many factors including the simple passage of time or being weakened through various mechanisms (see, for instance, "Weakening memories through remembering others"). In RealMemory, this is represented by running a MemMaintenance method telling the character how much world-time has passed. MemMaintenance tests each memory in the LTM and figures out whether it should be distorted or obfuscated. It's not the only way memories in the LTM change, however. Each time an NPC does something that might alter a memory (through memorizing something similar, like a new password, for example), this can affect other memories**.** Also, a character can temporarily overload his memory and be unable to remember everything about a certain subject/object/concept**,** such as trying to remember all the books on a bookshelf. These memories are not distorted or obfuscated; they are just temporarily unavailable.

Obfuscated memories can sometimes be retrieved through the surrounding context, be it a physical location or an emotional tie. See "That Reminds Me …" for more.

You can also associate stories with memories in the LTM. These are blocks of text that the NPC will remember wholesale, such as "Jeffrey used to have more hair." See "Using Only Stories**"** for more on this. In future versions we will be moving toward integration of these stories into the memory constructs themselves (and also into natural language capabilities), but for now they exist to give the developer more flexibility in creating memories. Stories provide a way for memories to have developer-created attributes, such as time (a past event) or opinion.

Of course, the developer is free to use only a subset of the capabilities of RealMemory. For example, the STM can be skipped, and memories can be made absolutely permanent and undistortable.

One use of RealMemory would be in a murder-mystery game, wherein each NPC might remember things differently.

INITIALIZING REALMEMORY

To use RealMemory in-game, you need to create an AIMemSensory object for each NPC. The AIMemSensory object acts as the interface between the NPC's "senses" and her mind. (If you're also using the Extreme AI Personality Engine, you'll see a similarity here.) In the variable declaration section of a script you're attaching to your NPC, declare something like:

[1] Note, however, that being similar may also distort the memory in the LTM if the information is contradictory.

```
public AIMemSensory myMem = new AIMemSensory();
```

You'll then need to call the Init method when the script is first run, in the Start method, for example. The simplest Init call is as follows:

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

where this.tag is the tag for the GameObject to which the script is attached, and myName is the name of the character (as has been set up in the Editor; see p. 19). You can create a new character at run-time by using the Init method as follows:

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

where true tells the method to create a new character (and also makes sure you didn't just make a typo). If you omit the Boolean and type in a non-existent character name, you'll get an error (three, actually: one for the LTM, one for the STM, and one cryptographic exception as it tries to decode a file that doesn't exist).

Note that this is also where you'd first tell RealMemory about your NPC's personality variables (or at least those that have an effect on human memory). In particular, these are:

Order
Self-Discipline
Depression
Trust
Positive Emotions
Competence
Anxiety

These are all facets of the Five Factor Model of personality. If you're using a different personality model, match as you see fit (using values from 0 to 100, with 50 being average and higher scores indicating a higher tendency toward the facet). If you don't provide any values, your NPC will be given average scores for all of these. Note that you can change these values at any time, as necessary during the course of an NPC's life (say, if his depression increases after his village burns down, or he becomes more disciplined after years in a Shao Lin monastery). See "How Do I Change Personality Values In-Game?"

For example, if you have floats representing each of these personality variables (obtained from a personality engine such as Extreme AI or determined in some other way), the Init call might look like:

```
myMem.Init (this.tag, myName, myOrder, myDiscipline,
myDepression, myTrust, myPosEmo,
        myCompetence, myAnxiety);
```

See the sidebar for more information.

Memories Based on Personality?

According to studies, several personality facets in the Five Factor model and associated moods and emotions can affect the encoding of memory in the STM and the strength with which it is transferred to the LTM. In particular, depression and anxiety interfere with the number of items and quality of information in the STM, and strong emotions generally affect the strength or weakness of encoding in both the STM and LTM. Further, optimism and self-assurance strengthen the STM, and there is evidence that order and self-discipline also affect the strength of the STM (represented here by the number of elements possible in the character's STM at any given moment). There are other possible connections, but many studies are contradictory or too vague.[1]

RealMemory can easily tie into the Extreme AI personality engine (full version). It can also be used with other personality or emotion systems.

Here's a sample Init using the Extreme AI Personality Engine to get the personality values:

```
void Start () {

        myAIChar.Init (this.tag,myName); //initializes Extreme AI

        //get the personality variables (using whichever engine or method is desired)
        //this example uses Extreme AI
        float myOrder = myAIChar.CharSingleFacetTool("Order");
        float myDiscipline = myAIChar.CharSingleFacetTool("Self-Discipline");
        float myDepression = myAIChar.CharSingleFacetTool("Depression");
        float myTrust = myAIChar.CharSingleFacetTool("Trust");
        float myPosEmo = myAIChar.CharSingleFacetTool("Positive Emotions");
        float myCompetence = myAIChar.CharSingleFacetTool("Competence");
        float myAnxiety = myAIChar.CharSingleFacetTool("Anxiety");

        myMem.Init (this.tag,myName,myOrder,myDiscipline,myDepression,myTrust,myPosEmo,
                myCompetence,myAnxiety);

}
```

HOW DO I CHANGE PERSONALITY VALUES IN-GAME?

During the course of a game, your NPCs are likely to change in personality (if using a personality engine), or you may decide that events have caused such a change (even if you're not using a personality engine). You can change the values that affect memory with the following methods:

ChangeMyOrder(float myNewOrder)
ChangeMyDiscipline(float myNewDiscipline)
ChangeMyDepression(float myNewDepression)
ChangeMyTrust(float myNewTrust)
ChangeMyPosEmo(float myNewPosEmo)
ChangeMyCompetence(float myNewCompetence)
ChangeMyAnxiety(float myNewAnxiety)

Use is simple; for example:

```
myMem.ChangeMyOrder(75f);
```

Values should be between 0 and 100.

In-Game Use of RealMemory: 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 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.

Using Only Stories

The story system in RealMemory allows a character to remember small blocks of information; for instance, an NPC (let's call him Jason) could remember the following:

Tilla Transit used to have green hair
The Fighter carries an anchor
The Healer hates fighting
The Bard is a terrible singer

Each story consists not only of the story text, but also its strength, any concepts related to the story, any emotional ties it may have for the NPC, and any "types," which is a special variable linking the story to a time, place, or anything else you wish to have as a category. For instance, with the story about Tilla Transit:

Story text:	Tilla Transit used to have green hair
Strength:	80 (can rate it between 1 (very, very weak, and likely to be forgotten) and 100 (very, very strong, unlikely to be forgotten)
Concepts:	Tilla Transit, green, hair
Emotion:	Shock (meaning that Jason is shocked that such a thing could have been true, NOT that Tilla is shocked by her own hair color)
Type:	Past (denoting that Tilla used to have this hair color, but Jason believes she doesn't now)

The primary difference between stories and the rest of the memory system is that stories are stored as complete sentences, whereas individual memories (in the LTM or STM) are concepts, or the sentences broken down into their component parts. Stories are not as flexible as individual memories, but are easier to use, and in fact can be used as a complete memory system if you'd like (this trades some of the realism for something perhaps a bit more game-friendly). In many ways, however, they are treated as real memories—they can be weakened and distorted over time, or forgotten (and potentially remembered in the right context). (Even in the stories-only mode, some parts of stories are placed in the LTM, which operates in the background; these parts are the Concepts that are input with a new story, such as "Tilla Transit," "green," and "hair" in the above example.)

Adding a Story to Memory

To add a story to an NPC's memory, use the AddWholeStory method:

> **AddWholeStory(string storyText, float storyStrength, string[] storyConcept, string[] storyType, string[] storyEmo)**—returns nothing, just adds a story to the character's memory.

For example, to store Tilla's hair story in Jason's memory:

```
myMem.AddWholeStory("Tilla Transit used to have green hair",80,new string[] {"Tilla Transit", "green", "hair"},new string[] {"past"},new string[] {"shock"});
```

Let's use the NPC's story about the Bard as another example:

Story text:	The Bard is a terrible singer
Strength:	50
Concepts:	Bard, singer
Emotion:	None
Type:	Opinion

In this case, the type has been marked as "Opinion"—that is, it's Jason's opinion that the Bard sings like a cat in the middle of a dark and rainy night (actually, that would be another story!). A character might take the type into account whenever asked to remember or retell a story; for instance, Jason could love telling only stories that are Opinions, or only stories that are the opinions of others (another type you could create), or in other words, gossip.

Adding this to Jason's memory:

```
myMem.AddWholeStory("The Bard is a terrible singer",50,new string[] {"Bard", "singer"},new string[] {"opinion"},new string[] {"none"});
```

Note that any field that has no value (in this case, emotion) should be filled with the word "none."

My Head Is Filled with Stories, but How Do I Get Them Out?

You can retrieve stories (or parts of stories) through the following methods:

MemRetAllStories(ref string[] memStories, bool interference = false)—returns (by ref) all the stories a character knows. Returns only the text of each story, not any of the associated concepts, types, etc.
StoryRetOnConcept(string[] stryConcept, ref string[] memStories, bool interference = false)—l; e.g., if the stryConcept array includes the strings "green" and "hair," the method will return only stories with one of those two concepts. Again returns only the text of the stories.
StoryRetOnType(string[] stryType, ref string[] memStories, bool interference = false)—returns (by ref) all the stories a character knows that include the type(s) included in the method; e.g., if the stryType array includes the string "opinion," the method will return only stories with that type. Again returns only the text of the stories.
StoryRetOnEmotion(string[] stryEmotion, ref string[] memStories, bool interference = false)—returns (by ref) all the stories a character knows that include the concept(s) included in the method; e.g., if the **stryEmotion** array includes the string "Angry," the method will return only stories with that emotion. Again returns only the text of the stories.
StoryRetOnStrength(float stryStrength, string relation, ref string[] memStories, bool interference = false)—returns (by ref) all the stories a character knows that are greater than, equal to, or less than stryStrength, according to the value of

Weakening Memories through Remembering Others?

According to studies, memories are not forgotten (lost completely from the LTM), but are sometimes distorted (in effect, overwritten) or lose all synaptic connection with the rest of one's memories. Interestingly, it has been found that the very act of remembering something can weaken other memories (Wimber et al, 2015; University of Birmingham, 2015); additionally, trying to remember similar memories (such as a list CDs) can create output interference, wherein the memories themselves are not weakened, but are interfered with (this is believed to relate to the amount of space in the STM/working memory). These are represented in RealMemory by two bool methods attached as appropriate to the various MemRet and StoryRet methods. By default, these are false; that is, no memories will be altered or temporarily forgotten. By sending "true" for changeOthers (not used with the Story methods, but used later in the LTM methods), the act of remembering one memory will weaken memories that have similar contexts, while strengthening the remembered memory. If you send "true" as the value of "interference", the character may begin to forget items in a long list, typically after reaching the limits of her working memory capacity. These items are not actually forgotten, nor are they weakened. The character's memory is just temporarily overwhelmed.

relation (which can be “>”, “=”, “<”, “>=” or “<=”). Again returns only the text of the stories.

For example, if you want to retrieve all the stories Jason knows, you’d type the following (assuming you have a string array named stryText set up to hold the story text):

```
MemRetAllStories(ref stryText);
```

Note that we’ve left out the interference Boolean. This means that, by default, the act of trying to remember these stories will not result in omitting some of the stories from the list.

I KNOW THE STORY ...

If you know the story text, the following methods will allow you to return other information about it.

StoryRetConcepts(string storyText, ref string[] storyConcepts, bool interference = false)—returns (by ref) all the concepts related to a specific story.
StoryRetTypes(string storyText, ref string[] storyTypes, bool interference = false)—returns (by ref) all the types related to a specific story.
StoryRetEmotions(string storyText, ref string[] storyEmo)—returns (by ref) all the emotions related to a specific story.
StoryRetStrength(string storyText)—returns (as a float) a story’s strength.

OTHER STORY-RELATED METHODS

Over time (or due to other issues, such as if the character has amnesia), an NPC can “forget” some memories (in this case, stories) or misremember them (see Memory Maintenance, below). Note that forgetting does not mean deleting; in line with prevalent theories, memories are not actually lost; they are instead obfuscated, losing their connection to normal means of recovery. However, if you wish to make a story permanent (not subject to obfuscation or distortion), or wish to make a permanent story into a non-permanent one, use the PermaToggleStories method:

PermaToggleStories(string storyText, bool changePerma)—changes a story’s permanence to the value in changePerma (true to make permanent, false to make non-permanent).

To find out whether a story is currently permanent, use StoryPermanent:

StoryPermanent (string memText)—returns whether a story is set as permanent or not (bool). Returns a bool.

Normally stories will automatically degrade to forgotten status through the use of MemMaintenance, but if you wish to change this status yourself, use ObfusToggleStory:

ObfusToggleStory(string storyText, bool changeObfus)—changes a story’s obfuscation to the value in changeObfus. True is obfuscated (forgotten), false is not.

THAT REMINDS ME ...

Have you ever remembered something just because of a smell (the rich chocolateness of baking cookies reminding you of an event from childhood, for instance) or finding an item (an old watch reminding you of when your grandfather gave it to you), even though you’d otherwise forgotten it? Current theory says that context (physical or emotional) plays a part in re-creating connections to an otherwise forgotten memory. In RealMemory, the developer can use contexts (stored as the concepts related to a story) or emotional associations (stored as

emotions related to a story) to test whether any obfuscated stories are remembered. For instance, over the years Jason has forgotten that Tilla's hair used to be green. But then he wanders into a wig shop and sees a green wig. We can test whether that sparks his memory using one of the following two methods:

TryObfuscatedStory(string thisContext)—returns the story text of an item that has been saved from obfuscation, given a single concept or emotional context. The obfuscated story with the greatest strength is the one that may be recovered (due to cognitive theories stating that memories with the same context are in competition with one another). Returns the string "nothing remembered" if nothing has been made un-obfuscated.
TryObfuscatedStory(string[] thisContext)—overload that returns the story text of an item that has been saved from obfuscation, given an array of concepts and/or emotional contexts. The obfuscated story with the greatest strength is the one that may be recovered (due to cognitive theories stating that memories with the same context are in competition with one another). Returns the string "nothing remembered" if nothing has been made un-obfuscated.

In Jason's case, we have two concepts to test ("hair" and "green"), so we'd use the second method:

```
string storyText = TryObfuscatedStory(new string[] {"hair", "green"});
```

Note that a story is not automatically remembered, but has a greater chance of being remembered the more contextual items (either concepts or emotions) match it.

The Years Go By: Memory Maintenance

In general, time will be the main enemy to a character's memory; the more time has passed (and the weaker the memory), the more likely that memory (in this case, a story) will be forgotten or distorted in some way (e.g., Tilla's hair color will be misremembered as blue).

In RealMemory, you can set the number of days (or years, or whatever) between memory maintenance updates, making sure that storekeeper no one has visited in a year doesn't remember the characters like it was yesterday (well, unless that's appropriate for some reason). You can call MemMaintenance whenever necessary before such an encounter. This is done by setting ChangeCritTime:

ChangeCritTime(float daytime)—returns nothing, but sets the amount of world time that must pass between maintenance updates.
CheckCritTime()—returns the number of days currently set as the amount of time that must pass between maintenance updates.

MemMaintenance(int dayTime)—returns nothing, but checks whether a character's memory is due for updating. If accumulated dayTime is greater than a critical value that you set elsewhere (in ChangeCritTime), maintenance occurs.

MemMaintenance lowers non-emotionally charged story strengths by a preset amount (defaulting to 1 per check, but this can be changed using **ChangeLossOverTime(float loss)** [you can check the current loss using **GetLossOverTime()**]). It also checks whether any memories have weakened sufficiently that they might be distorted (e.g., a property of the memory, such as color, changed from one value to another) or even obfuscated (marked as "forgotten" and irretrievable under normal circumstances). As described previously, an obfuscated memory is not deleted; it can be eventually retrieved through extraordinary means, such as the character being in a similar context to that she was in during the memory's encoding (this can be a place, smell, emotional state, etc.). To attempt to retrieve such a memory, use the method **TryObfuscatedMem**.

Distorted memories can be "fixed" only by new input that corrects the character's misperception.

MemMaintenance has less effect on emotionally charged memories and no effect on memories you've marked as "Permanent."

Saving Stories

Stories must be saved to the appropriate xml file when a player quits the game or saves at a save point. This is easily done through the **SaveStories()** method. Note that only the data for a single character is saved—for instance, if using the Jason example we've been using, myMem.SaveStories() would save only Jason's memories.

That's it! If you're using only stories, and don't need to add any stories to characters before the game starts, you're set! If you need to add stories to characters before gameplay, you can use the RealMemory Editing tools in the Tools menu (see Creating Memories with the Editor). If you'd like a bit more realism in your memory system, use the long-term memory system, as described below!

Using the LTM

For a more realistic memory system, in addition to using stories, you can utilize your NPC's long-term memory (LTM). In RealMemory, a character's LTM contains memories that are each their own object—in a way, the same as a specific pattern of neuronal firing that represents a memory in our own minds (as far as we understand it at present). These memory objects have related concepts that tie them to other memories in a loosely hierarchical way, both to subconcepts and to higher (or ur) concepts. Think of it like this: Jason knows Tilla Transit (a memory), and he knows Tilla is a person—or, rather, a member of the higher (ur) concept "person." He also knows that Tilla has eyes (related concept), and that her eyes are blue (property of the related concept) and round (another property). He also knows that people in general have eyes (related concept of the ur concept "person") and can guess that all people have eyes; further, he knows that eyes have color (an ur concept; color contains all the specific colors Jason knows) and shape (another ur concept).

When a new memory is created in the LTM (either directly or arriving from the STM), it can strengthen existing concepts (for example, confirming that eyes are part of the concept person) or add new ones (such as adding lilac to Jason's internal list of colors). It can also create a situation in which Jason's memories are confused or distorted—say he remembers Tilla's eyes as being blue, but another friend states that they are lilac—and his LTM will determine which of these he will believe going forward.

During gameplay, you can add new concepts directly to an NPC's LTM[2] using the **AddToLTM** method:

AddToLTM(string memText, float memStrength, List<MemConceptGroup> memConcepts, List<ContextItems> memContextItems, string memEmotion = "none", bool makeUr = false)—This method is complicated, so let's break it down:

- memText—This is the memory itself, typically very short, such as a person's name (representing the person himself), an object (a knife), or the like. This is a string variable.
- memStrength—Ranges from 0-100, with higher strength meaning the memory is stronger and less subject to distortion or obfuscation. (Note that the STM, if used, determines this value from environmental, personality, and emotional factors.)
- memConcepts—This represents any subconcepts related to this memory and their properties. A person (Jeffrey, say) might(!) have hair, eyes, and skin, with properties of hair, for instance, being length (long)

[2] This is, of course, not exactly how humans add memories; we add potential memories to our short-term memory, from whence the memory might make it to our LTM. Thus, for the most realism in RealMemory, use the short-term memory system (see Real RealMemory: Using the Short-term Memory).

and color (blond). Each subconcept is an object of class MemConceptGroup, and memConcepts is a list made up of these (List<MemConceptGroup>). See below for an example.

- memContextItems—This includes any context that the character might associate with this memory later, such as seeing a haunted house reminding you of the murder that took place there, the smell of hot dogs reminding you of a baseball game, and so on. Each such context item is an object of class contextItems, and memContextItems is a list made up of these (List<contextItems>). contextItems include a string (the context itself) and a float (the strength of the context). See below for an example.
- memEmotion—The emotion (if any) with which this memory is associated. Set to "none" if there is no emotive resonance.
- makeUr—Tells the character whether a memory is an overarching category like "color" or "shape." Used only if you're teaching your character very basic concepts; usually she will sort it out through the "propUr" values in properties (see memConcepts, above, or the section on using the STM.

An Example

Say the NPC, Jason, is witness to a murder. It's dark, the fog's rolling in off the moors, but he sees a few things that he will remember later. For instance, there's the knife, which he may remember in some detail. How do we pass this information to his LTM?

First, we know that memText is going to be the knife itself, in this case "murder knife." And Jason remembers it well; its memStrength is going to be 80.

The most complicated part of the method call is the memConcepts list. We know the memory is going to be "murder knife," but what things about the knife does Jason notice?

The list is made of up MemConceptGroup objects. Each such object includes the following:

- memConcept—The related concept itself. This also becomes a memory in its own right, if the concept does not already exist in the NPC's memory. A knife's handle and blade might be two memConcepts associated with the memory "murder knife."
- memConStrength—The strength of the connection between the memConcept and the original memory. Ranges from 0 to 100, and will default to 50 if not specified. Used if for some reason the concept is loosely or strongly tied to the original memory; a lower number has a greater chance of being misremembered later.
- memProperties—A list of properties of the memConcept. This includes:
  - propName—The name of the property. If the blade were bloody, long, and sharp, these would be propNames. (Whether Jason could really tell the knife was "sharp" is up to you; you have to act as his senses.)
  - propStrength—The strength of the property memory, ranging from 0 to 100. Again, the lower the property strength, the more likely it will be forgotten or misremembered (for example, if Jason only barely remembers the length of the knife's blade with a strength of 10, he may later misremember it, thinking it was short instead of long). Defaults to 50.
  - propUr—The overarching category to which this property belongs. "Bloody" might belong to the overarching (ur) concept "cleanliness." Similarly, "long" would be part of the category "length." See Ur Concepts/Categories for more detail.

In Jason's case we could create a set of related concepts as follows (in c#):

```
List<memConceptGroup> memConcepts = new List<memConceptGroup>();

memConcepts.Insert (0, new memConceptGroup{memConcept="blade",memConStrength=100});
memConcepts [0].memProperties.Add (new properties {propName="sharp",propUr="sharpness"});
```

```
memConcepts [0].memProperties.Add (new properties {propName="long",propUr="length"});
memConcepts [0].memProperties.Add (new properties {propName="bloody",propUr="cleanliness"});
```

Note that we're letting the property strengths default to 50. Note also that in this case you're inserting the new related concept at position 0 in the first line, and then adding the properties one by one. This can be done all on one programmatic line:

```
memConcepts.Add (new memConceptGroup{memConcept="blade",memConStrength=100, memProperties=new List<properties>{new properties{propName="sharp" ,propUr="sharpness"}, new properties{propName="long" ,propUr="length"},new properties{propName="bloody" ,propUr="cleanliness"}}});
```

Another list you need to send is the memContextItems list. It's much simpler than the first list, 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. Let's say that the context of this memory is Tilla's house. You can create this list (in this case, a short list of only one item) as follows:

```
List<contextItems> thisContext = new List<contextItems>();
thisContext.Add (new contextItems{contextName="Tilla's house",contextStrength=80});
```

We then need to know the emotional context, if any. This is similar to the emotional context of stories (described in the previous section). In this case we'll say "fear" is an appropriate emotional context for this memory.

To then add the entire memory into the LTM you'd add a line calling AddToLTM:

```
myMem.AddToLTM ("murder knife", 80, memConcepts, thisContext, "fear");
```

Note that we're not making "murder knife" an overarching concept, so we let it default to makeUr = false.

Another Example

Say you did want to create an overarching concept for the NPC to grapple with (and wanted to do it during gameplay, rather than in the Editor). What then? Well, let's give Jason an idea of what "color" is.

Again, memText and memStrength are easy: memText = "color" and memStrength = 100 (because we want Jason to definitely remember what colors are). And, in this case, the related concepts (memConcepts) are easier, too, as they have no properties (they are all just colors):

```
List<memConceptGroup> memConcepts = new List<memConceptGroup>();
memConcepts.Insert (0, new memConceptGroup{memConcept="green",memConStrength=100});
memConcepts.Insert (0, new memConceptGroup{memConcept="blue",memConStrength=100});
memConcepts.Insert (0, new memConceptGroup{memConcept="yellow",memConStrength=100});
memConcepts.Insert (0, new memConceptGroup{memConcept="grey",memConStrength=100});
```

There is no context for these, so we'll just create an empty contextItems object:

```
List<contextItems> thisContext = new List<contextItems>();
```

There's no emotional context, either, so the AddToLTM call is:

```
myMem.AddToLTM ("color", 100, memConcepts, thisContext, "none", true);
```

Can’t this @$#% method call be made any simpler?

Well … no and yes. No in the sense that, to make the memories function in as realistic a way as possible, none of the parameters can be removed. However, we’ve created a similar **AddNewSimpleMemory** method that requires only the memory and its strength (from 0-100):

AddNewSimpleMemory(string newMemory, float memStrength, string memEmotion = "none", bool makeUr = false)—This adds a memory with no related concepts and no context (a simple memory indeed!).

Leaving a memory isolated like this is not conducive to realism, but can be used if your gameplay needs only this level of memory. Isolated memories can still undergo maintenance and become obfuscated, but typically wouldn’t be distorted. You can, however, convert a simple memory to a more complex one through adding related concepts and contexts to it:

AddRelConToMem(string memText, List<MemConceptGroup> memConcepts, string memEmotion = "none")—Adds a related concept (only one) to an existing memory. As before, memConcepts are any subconcepts related to this memory and their properties (see above).

ChangePropertyInMem(string memText, string relConText, string oldProp, List<Properties> memProps)—Changes the value of a property of a related concept in a memory; for instance, you could change the color of Tilla’s hair from brown to blonde. This could be done as follows:

```
myMem.ChangePropertyInMem(“Tilla Transit”, “hair”, ”brown”, new List<Properties>{new Properties{propName=”blonde”, propStrength=50}});
```

Adding Other Elements to a Memory after It’s Formed in the LTM

As the developer and ultimate arbiter of your characters’ memories, you can go into a memory after it has been created and add to or delete elements from it. The following methods show you how to do this.

Adding Context

As with other elements of the character’s memory, you can add context items after the memory has been created in the LTM. Again, context includes the kinds of things that might remind you of a memory, such as seeing a haunted house reminding you of the murders that took place there, or smelling hot dogs reminding you of baseball, or hearing a babbling brook reminding you of that lovely summer hike you took after university was finished.

You can add or delete context by calling the following methods:

AddMemContext(string memText, string memContextName, float memContextStr = 50)—creates a new context (memContextName) for the memory memText, with a default strength of 50 (and ranging from 0-100).

DeleteMemContext(string memText, string memContextName)—deletes a context item from a specific memory, if it exists.

CheckContext(string memText, string contextToCheck)—returns bool telling you whether the a context item exists in a specific memory.

Conflicting information

Sometimes a character will encounter information that differs from that which he thinks he knows, such as learning that a character's eyes are lilac, not blue, or being told that a suspect is blond-haired, not brown-haired. Whether this information changes the NPC's idea of the hair color is another thing, depending on the trustworthiness of the source of the info and the strength of the character's thoughts on the subject.

If a character has no specific knowledge of the information (e.g., has no idea what color the suspect's hair might be), or has only a weak idea of it (e.g., kinda thinks it was brown, but isn't sure), then the new information will likely override the old.

One interesting use of this is the application of the "malleability of memory" (The Human Memory: Memory Consolidation), which includes instances of people constructing memories after the fact, as when a police officer shows a "picture of a single individual to a victim and ask if the victim recognizes the assailant. If the victim is then presented with a line-up and picks out the individual whose picture the victim had been shown, there is no real way of knowing whether the victim is actually remembering the assailant or just the picture." In RealMemory, if the character has no prior knowledge of the assailant, or perhaps a vague recollection, and trusts the officer, there's a high likelihood of the character's memory being changed. Note that, as you have to be the character's senses, you'll have to decide how to send this information to the STM or LTM, and how strongly to encode it. (You could also tie this in to a personality call; for instance, if the NPC is really trusting generally, and the officer is very assertive, then you could make the new, "incorrect" memory very strong.) The internal LTM process will take care of determining which memory stays in the NPC's LTM.

Asking a Character Questions: Retrieving Memories from the LTM

So you've sat Jason down and are asking him about the murder he witnessed. "What," you ask, "about the murder knife? What did it look like?"

There are several methods you can use to retrieve memories, as described briefly below.

MemRetDoIKnow (string memText, bool changeOthers = false)—returns a Boolean value, checking only that the character knows a concept. This is the simplest memory query. For instance, does Jason know about a "murder knife"? Yes, he does. But we wouldn't know anything else he knows or thinks about the knife.

MemRetAllConcepts (ref string[] allConcepts, bool interference = false)—returns (by ref) all memories in the LTM. Returns names of memories only.

MemRetAllNonUrConcepts (ref string[] allNonUrConcepts, bool interference = false)—returns (by ref) only those memories that are not overarching categories. In Jason's case, this could return Tilla Transit, long, murder knife, blue, lilac, and round, but not person, weapon, color, or shape.

MemRetAllUrConcepts (ref string[] allUrConcepts, bool interference = false)—returns (by ref) only the overarching categories by which the character organizes things. Just the opposite of the previous method: would return person, weapon, color, and shape, but not the rest in that list.

CheckIfUrQuick(string memText)—checks whether a memory is an ur concept/category. Returns a Boolean. For instance, would return true if asked about person, but false if asked about Till Transit.

MemRetEmoTie(string memText)—returns any emotional charge the memory has (as a string). Asking Jason about a "murder knife" would return "fear".

MemContext(string memText, ref string[] memContext, ref float[] memContextStr)—returns (by ref) any contexts that tie in to the memory, such as a location. This can help in remembering obfuscated ("forgotten") memories. See the next section, XXX, for more information.

MemRetOneKnown (string oneInfo, string aboutX, ref string[] propsAboutInfo, ref float[] propsAboutStrength, bool interference = false, bool changeOthers = false)—returns a string telling you whether a character knows a related concept about a memory. Also returns by ref an array of property names and an array of matching strengths of that related concept. Does Jason know that Tilla Transit has eyes? Yes. What does he know about them? They are blue and round.

MemRetInCat (string oneInfo, string inThisCat, bool changeOthers = false)—returns a Boolean value, checking whether a memory is in a specified ur category, such as whether "green" is part of the category "color."

MemRetInAnyCat (string oneInfo, ref string[] anyCats, ref float[] anyStrength, bool interference = false, bool changeOthers = false)—returns (by ref only) the catgory (or categories) in which a memory can be found, and any associated strength of connection to that category. Is "blue" in a category? Yes, "color".

MemRetWhatTypes (string xGenItem, ref string[] typesAboutX, ref float[] typeStrength, bool interference = false, bool changeOthers = false)—returns (as refs) the kinds of things the character knows about a regular memory—basically the related concepts, but without the properties and without any ur categories. Asking Jason about Tilla Transit would return eyes, but not person. Won't return any info about an ur category.

MemRetWhatTypesBoth (string xGenItem, ref string[] typesAboutX, ref float[] typeStrength, bool interference = false, bool changeOthers = false)—returns (by ref) not only the related concepts of regular memories but also the related concepts of ur categories (such as that people have eyes, hair, and skin). Does not return the members of an ur category (it would not tell you that Tilla Transit is a member of "person").

MemRetTypesTwo (string xGenItem, ref string[] typesAboutX, ref float[] typeStrength, bool interference = false, bool changeOthers = false)—returns (by ref) only the related concepts of ur categories, but doesn't work on regular memories.

MemRetTypesPlusUr (string xGenItem, ref string[] typesAboutX, ref float[] typeStrength, bool interference = false, bool changeOthers = false)—returns (by ref) all related concepts and members of an ur category; e.g., asking this about "person" might return all the general related concepts "hair", "eyes", and "skin" and also the specific members of the category, such as "Tilla Transit".

MemRetWhatInUr (string xGenItem, ref string[] typesAboutX, ref float[] typeStrength, bool interference = false, bool changeOthers = false)—returns (by ref) all the members of an ur category, such as all the members of "color" (say, "red","green", and "blue").

What do the bool values at the ends of many of these methods do? See the sidebar on pg. 6 for info about weakening and interfering with memories using "interference" and "changeOthers" variables.

Memory Maintenance (and distortion and obfuscation)

As characters live their lives, some memories fade or become changed over time. In RealMemory, you can set the number of days (or years, or whatever) between memory maintenance checks, making sure that storekeeper no one has visited in a year doesn't remember the characters like it was yesterday (well, unless that's appropriate for some reason). You can call MemMaintenance whenever necessary before such an encounter.

MemMaintenance(int dayTime)—returns nothing, but checks whether a character's memory is due for updating. If accumulated dayTime is greater than a critical value that you set elsewhere (in ChangeCritTime), maintenance occurs.

MemMaintenance can lower memory strengths in the LTM by a preset amount (if you want a character's memories to fade over time; this isn't particularly realistic according to most modern theories, except in the case of disease or injury, but for game purposes it's allowed). It also checks whether any memories have weakened sufficiently that they might be distorted (e.g., a property of the memory, such as color, changed from one value to another) or even obfuscated (marked as "forgotten" and irretrievable under normal circumstances). An obfuscated memory is not deleted; it can be eventually retrieved through extraordinary means, such as the character being in a similar context to that she was in during the memory's encoding (this can be a place, smell, emotional state, etc.). To attempt to retrieve such a memory, use the method **TryObfuscatedMem**.

Distorted memories can be "fixed" only by new input that corrects the character's misperception.

MemMaintenance has less effect on emotionally charged memories and no effect on memories you've marked as "Permanent."

Other LTM-related methods

MemPermanent (string memText)—returns whether a memory is set as permanent or not (bool).
PermaToggle(string memText, bool changePerma)—changes a memory's permanence to the value in changePerma.
ObfusToggle(string memText, bool changeObfus)—changes a memory's obfuscation to the value in changeObfus.
TryObfuscatedMem(string thisContext)—returns the memory text of an item that has been saved from obfuscation, given a single concept or emotional context. Returns the string "nothing remembered" if nothing has being made un-obfuscated.
TryObfuscatedMem(string[] thisContext)—overload that returns the memory text of an item that has been saved from obfuscation, given an array of concepts and/or emotional contexts. Returns the string "nothing remembered" if nothing has being made un-obfuscated.

Saving Memories

Just as with stories, you'll need to save the NPC's LTM whenever the game is saved. And just as with stories, this requires a simple method call: **SaveLTM()**. For example, in Jason's case, we'd use myMem.SaveLTM(). Again, this saves only one character's memories; you need to call this method for each NPC.

Real RealMemory: Using the Short-Term Memory (and LTM and Stories)

To make your NPCs as human-realistic as possible, you can utilize their short-term memories (STMs) as well. This gives them pretty much everything a human memory system has. 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; described previously).

Sensory Memory

The one thing we are unable to mimic with this system is the sensory memory, meaning the initial input via sight, hearing, touch, taste, smell, etc. This has to be provided by you, the developer. If the character witnesses a

murder, you will have to, in effect, tell her all the things she's seen, heard, etc. This is done through the following method:

SensoryMemory(string memoryItem, string senseType, List<MemConceptGroup> memConcept, List<ContextItems> memContextItems, int attentionLvl = 0, int environsLvl = 0, int motivationLvl = 0, string fromChara = "", int emoCharged = 0, string emoSaved = "none")

Or in slightly more descriptive pseudocode:

SensoryMemory (memory, typeOfInput, relatedConceptsAndProperties, anyContext, attentionLevel, environmentLevel, motivationLevel, sourceOfMemory, emotionalCharge, associatedEmotion);

Nice and complicated, that! But very similar to the LTM method **AddToLTM** discussed earlier. Let's break it down …

- memory—This is the memory itself, typically very short, such as a person's name (representing the person himself), an object (a knife), or the like. This is a string variable.
- typeOfInput—This is the "sense" being used: this can be auditory, visual, smell, taste, touch, or a combination of these. These were chosen because they have a demonstrable effect on the strength of the input memory and whether it has a better chance of moving to the LTM. This is also a string. Strings you can enter include those in the table below (in combinations, A=auditory, V=visual, S=smell, Ta=taste, and To=Touch):

○ visual	○ bothAV	○ threeAVS	○ fourAVSTa	○ allFive
○ auditory	○ bothAS	○ threeAVTa	○ fourAVSTo	
○ smell	○ bothATa	○ threeAVTo	○ fourAVTaTo	
○ taste	○ bothATo	○ threeASTo	○ fourASTaTo	
○ touch	○ bothVS	○ threeASTa	○ fourVSTaTo	
	○ bothVTa	○ threeATaTo		
	○ bothVto	○ threeVSTo		
	○ bothSTa	○ threeVSTa		
	○ bothSto	○ threeVTaTo		
	○ bothTaTo	○ threeSTaTo		

- relatedConceptsAndProperties—This represents any subconcepts related to this memory and their properties. A person (Jeffrey, say) might(!) have hair, eyes, and skin, with properties of hair, for instance, being length (long) and color (blond). Each subconcept is an object of class MemConceptGroup, and relatedConceptsAndProperties is a list made up of these (List<MemConceptGroup>).
- anyContext—This includes any context that the character might associate with this memory later, such as seeing a haunted house reminding you of the murder that took place there, the smell of hot dogs reminding you of a baseball game, and so on. Each such context item is an object of class contextItems, and anyContext is a list made up of these (List<contextItems>). contextItems include a string (the context itself) and a float (the strength of the context).

The other parameters (not in the LTM version) are necessary to tell the NPC how strong the incoming memory is:

- attentionLevel—The amount of attention the character is paying to the potential memory. Ranges from -2 (not paying any attention at all) to 2 (really able to focus), with 0 being average.
- environmentLevel—Whether the surrounding environment is calm or distracting. Again ranges from -2 (very noisy and distracting) to 2 (very calm), with 0 being average.
- motivationLevel—The character's motivation to learn this piece of information. Also ranges from -2 (highly unmotivated) to 2 (very motivated), with 0 being average.

- sourceOfMemory—The person, etc that is the source of the memory (e.g., telling the NPC about something). Used to determine whether the NPC thinks the info came from a trusted (or untrustworthy) source.
- emotionalCharge—The amount of emotional charge associated with a memory. Can be negative, as a high emotional charge in one object tends to make surrounding objects less memorable (as in attention narrowing and the "weapon focus effect," where "witnesses to a crime tend to remember the gun or knife in great detail, but not other more peripheral details such as the perpetrator's clothing or vehicle)" (The Human Memory website). Ranges from -2 (very diminished effect) to 2 (highly charged effect).
- associatedEmotion—The specific emotion associated with the memory. This can become part of the memory's context, and the NPC may later recall this memory in a similar emotional situation.

AN EXAMPLE

Let's return to our favorite NPC, Jason, who is witness to a murder (just as in the LTM example earlier). As you'll recall, it's dark, the fog's rolling in off the moors, but he sees a few things that he may or may not remember later. For instance, there's the glint of the knife, which he may remember in some detail. This time, rather than plopping these events directly into his LTM, we're letting Jason's mind act as ours do: we'll provide him with the sensory information (what he sees, in this case), and he'll store this information temporarily in his STM (and some of it will get all the way to his LTM). How do we pass this information to his STM?

The most complicated part of the method call is the relatedConceptsAndProperties list. We know the memory is going to be "murder knife," but what things about the knife does Jason notice?
The list is made of up memConceptGroup objects. Each memConceptGroup includes the following:

- memConcept—The related concept itself. This also becomes a memory in its own right, if the concept does not already exist in the NPC's memory. A knife's handle and blade might be two memConcepts associated with the memory "murder knife."
- memConStrength—The strength of the connection between the memConcept and the original memory. Ranges from 0 to 100, and will default to 50 if not specified. Used if for some reason the concept is loosely or strongly tied to the original memory; a lower number has a greater chance of being misremembered later.
- memProperties—A list of properties of the memConcept. This includes:
    - propName—The name of the property. If the blade were bloody, long, and sharp, these would be propNames. (Whether Jason could really tell the knife was "sharp" is up to you; you have to act as his senses.)
    - propStrength—The strength of the property memory, ranging from 0 to 100. Again, the lower the property strength, the more likely it will be forgotten or misremembered. Defaults to 50.
    - propUr—The overarching category to which this property belongs. "Bloody" might belong to the overarching (ur) concept "cleanliness." Similarly, "long" would be part of the category "length." See The Concept of Concepts for more detail.

In Jason's case we could create a set of related concepts as follows (in c#):

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

relatedConcepts.Insert (0, new MemConceptGroup{memConcept="blade",memConStrength=100});
relatedConcepts [0].memProperties.Add (new properties {propName="sharp",propUr="sharpness"});
relatedConcepts [0].memProperties.Add (new properties {propName="long",propUr="length"});
relatedConcepts [0].memProperties.Add (new properties {propName="bloody",propUr="cleanliness"});
```

Note that in this case you're inserting the new related concept at position 0 in the first line, and then adding the properties one by one. This can be done all on one programmatic line:

```
relatedConcepts.Add (new memConceptGroup{memConcept="blade",memConStrength=100, memProperties=new List<properties>{new properties{propName="sharp" ,propUr="sharpness"}, new properties{propName="long" ,propUr="length"},new properties{propName="bloody" ,propUr="cleanliness"}}});
```

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="Tilla's house",contextStrength=80});
```

So far, this is almost exactly the same as the AddToLTM call. But then we have the other parameters:

senseType: We skipped this one to discuss all the more complicated parameters. Jason is seeing, but not smelling or hearing the murder knife. We'll go with "visual" only.
attentionLevel: He's definitely paying attention! Let's go with a 2 (really focused).
environmentLevel: With the fog and all, let's say the environment is a little difficult. A -1 for this parameter.
motivationLevel: Jason is motivated to try to remember this. A 2 for this one.
sourceOfMemory: There is no person or such telling this to Jason; he's seeing it for himself. We'll just send an empty string "".
emotionalCharge: This is the main object on which Jason is focusing, so it gets a 2. Note that other items in the scene (say, a lamppost nearby) would then get a negative emotionalCharge because they are peripheral details.
associatedEmotion: This gives an emotional context for this memory. In this case, "fear."

To then add the entire memory into the STM you'd add a line calling the method SensoryMemory:

```
myMem.SensoryMemory ("murder knife", "visual", relatedConcepts, thisContext, 1, 0, 0, "", 1, "fear");
```

CAN'T THIS @$#% METHOD CALL BE MADE ANY SIMPLER?

Well … no and yes (as with **AddToLTM**). No in the sense that, to make the memories function in as realistic a way as possible, none of the parameters can be removed (in fact, we could probably add a few). However, the only absolutely required information is the memory item itself, the sense type, the related concepts and properties, and the context list. Even with these you could send an empty list of related concepts and an empty list of context items.

In fact, we've created a similar **AddNewSimpleMemory** method that requires only the memory and its strength (from 0-100). (This requires skipping the STM entirely, so you're sacrificing realism of course. The strength must be included because normally the STM handles that variable. For more on this, see the section "Using Only the LTM."

WHAT HAPPENS INSIDE THE STM?

Once the memory is sent off to the character's STM, the memory's overall strength is calculated using the information you provided in the method call (e.g., environment, emotional tie) and other information based on the way personality and mood affect memory retention, whether this memory reinforces similar memories or similar related concepts, and whether this is a reinforcement of another memory already in the STM (such as when you try to memorize a date, then repeat it to yourself a few seconds later). If the overall memory strength reaches a certain critical level, it is transferred to the LTM and dropped from the STM. If it hasn't reached this level, it will

remain in the STM until it is either forced out by competing memories in the STM or until it decays in strength to zero and is dropped from the STM.

Some things will nearly guarantee transfer to the LTM, including strong emotional influence and reinforcing the memory once or twice (by, for instance, sending the same memory a second or even third time, mimicking the NPC trying to remember something by repetition). You can also skip the STM altogether if, for game purposes, you absolutely must have a memory in the LTM.

It's Taking So Long to Empty the STM …

Humans are able to keep things in short-term memory for varying amounts of time, but on average a memory stays 10-15 seconds before disappearing (unless it's moved on to the LTM). RealMemory operates by default much faster than this (memories stay in the STM on average around 4 seconds), but even this won't seem fast enough in some instances (for instance, if you're preloading the STM before the game starts; say, if you want the NPCs to "observe" a murder before the player gets there to question them). You can speed things up (or slow things down) using:

SecsBetChecksOverride(float numSecs)

You can check the current value by using:

float GetSecsBetChecks()

See, as an example, the Quick(ish) Start Guide.

Creating Memories with the Editor

The Concept of Concepts

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.

These relationships are represented in the Memories window in a few ways. Perhaps the best thing to do is to demonstrate.

Say you have a memory concept named "Jeffrey." If you've selected this concept, the screen will read "Jeffrey has", followed by a list of related concepts. Related concepts are those that are part of the memory above them—so Jeffrey has "eyes." These related concepts may have general properties that are displayed in the next popup—eyes have color, and so Jeffrey's eyes have the property "green."

Of course, we're building characters from the ground up, so we first have to teach the character about color, and green, and eyes in the first place (although in this example these things came with your download of RealMemory, so you don't have to teach the NPC *everything*).

These leads us to one important distinction between concepts, which you can see in the layout of the Memories window—memory concepts (in the top half of the window) and ur concepts (in the bottom half). Ur concepts are general concepts such as color that contain specific members of that concept—sort of like (in programming) a class object being a general concept, and instantiated versions of that class being specifics. Or think of it as categories: color is a category, and green, blue, red, etc. are members of this category.

Using the Editor (Creating Characters Prior to Gameplay) (Advanced Editing)

This gets into the nuts and bolts of memory creation, using the Unity Editor to create new characters and edit existing ones. Go to Tools->RealMemory to access the submenus discussed below.

A note on setting up characters

Actually, character setup is easy—you just type in a new character name in any of the submenus, then begin editing! When a character is created, two xml files will show up in ExtremeAI/Resources/MemTables, one for the character's memories and one for her stories. Thereafter, the character will be available from a dropdown list at the top of each menu (or you can type in a new name to create another character). A character begins with no memories and no stories.

Deleting a Character

Deleting a character is also easy. Go to the RealMemory/Resources/MemTables folder and delete the two files having to do with the character: xmltblLTM + character name and xmltblStories + character name.

ADDING MEMORIES THE EASY WAY (MEMORY PACKS)

With RealMemory, you can buy memory packs that automatically create memories, related concepts, ur-concepts, and properties without the hassle of adding each one on your own. In fact, RealMemory comes with a mempack ready to install, MemPackDemo.xml.

To install memory packs, place the memory pack file in the ExtremeAI->Resources->MemPacks folder (the Demo memory pack is already there). Then go to the Tools Menu in the Unity Editor, go to the RealMemory submenu, and choose Install Memory Pack. 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). After you've made these choices, click the Install Memories! button. If you wish to install more memory packs, go through the same process again; if you're done, click Done to exit the window.

Note that we can make custom memory packs for your particular game environment for a small fee, and that we will be coming out with more memory packs in the future.

ADDING INDIVIDUAL MEMORIES: THE MEMORIES WINDOW

Of course, there will be times you'll want to create your own memories, and do so prior to gameplay itself. To create a typical memory, go to the Memories submenu of RealMemory. A window similar to the one in Figure 1 will show up.

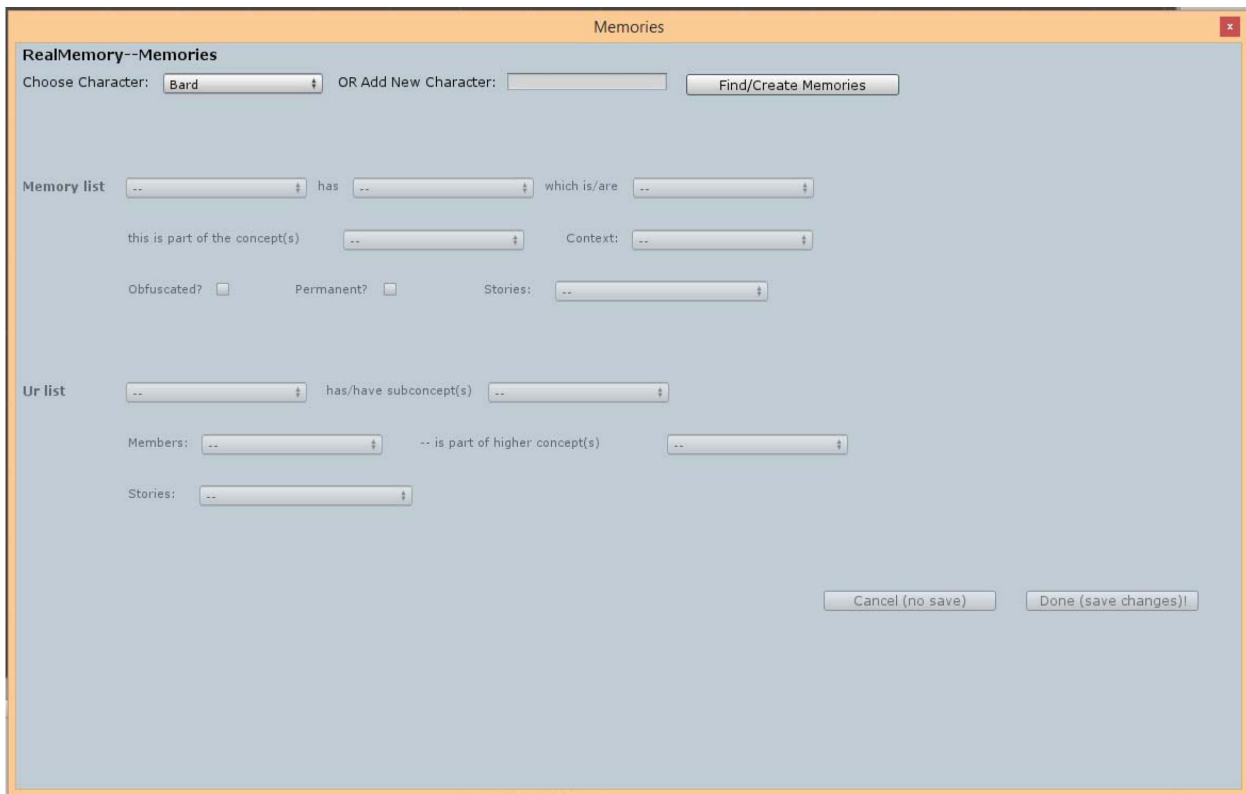

FIGURE 1

Choose character, or add new character

The top row allows you to choose from an existing character from a drop-down list, or to type in the name of a new character in the text box. Then click the Find/Create Memories button to bring up the character's current memories.

Note that if there is any text in the text field, RealMemory will assume that you want to create a new character with that name. (And if you type in the name of an existing character, it will retrieve that instead.)

I have all these concepts … now what?

Take a look at the Memory window. You'll see it's divided into two parts: the memory list and the ur list.

The Memory list includes all non-ur memories. To add (or edit) one, click Add/Chg Mem. This will bring up the following window:

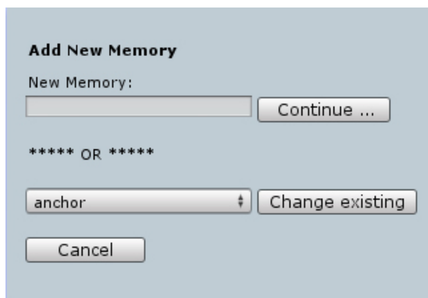

FIGURE 2

Note that you can add multiple memories at once by putting a comma between each one (no space!), as follows: red,blue,green

Whether you choose an existing memory or type in a new one, you will be able to edit its strength and emotional tie. Click Done to save it to the list.

To delete a memory, just click Del Mem, which will bring up a dialogue box confirming the memory you wish to delete.

DO NOT add any ur concepts to the regular memories list! These are added in the bottom half of the window. If you find you have an ur category in your regular memories list, you'll need to delete it for everything to work properly.

A memory has (or can have) various related concepts, and thst is what you enter in the next box (to the right of the memory itself, after the word "has"). For instance, the Fighter has a memory of his anchor, and his anchor has weight. Or as another example, check out the Fighter's memory of "Healer" (a specific character of that name, one of his pals(?) in our game SteamSaga). The Healer has hair, eyes, and several other relationships. These related concepts are generally things you could say the main memory "has": and anchor has weight, Healer has hair. If the Fighter knew Tilla Transit, he would think of her as having hair as well. You would not, however, say that the anchor has heavy, or Healer has red; these are properties of the related concepts.

Some related concepts are, in fact, ur catagories; weight would be one of these, as would height. They have properties that are related directly to themselves; that is, they have no subcategories. Other related concepts are regular memory concepts that have their own related concepts, like hair. Hair has the related (ur) concepts color and length.

Let's add a new related concept to a memory. (Actually, if you want to follow along, you'll have to delete a concept first; delete weight from anchor using the Del Conc button.) You'll get a popup window similar to the following:

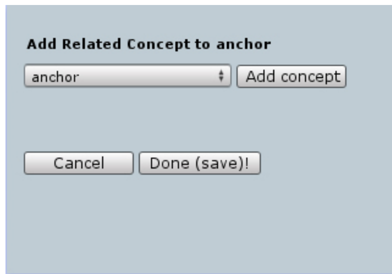

FIGURE 3

You can choose from any of the available memories, including ur concepts. To add weight to anchor, click on weight and click the Add Concept button. It will pop up with a list of properties available for weight. Go ahead and choose one; if you choose "none", the Fighter will have no memory of how heavy the anchor is; he'll just know it has weight. Make sure and click Done (save) when you're ready, or the concept won't be added.

Some related concepts (like hair) have multiple properties, all of which will show up in the Add Related Concepts box.

HOW DOES IT KNOW THAT HAIR HAS THESE PROPERTIES?

It doesn't, at least not without you teaching it. Hair at some point needed to be entered as a new memory concept, and it was given the related concepts color and length. If you're adding related concepts that need to remain general (you wouldn't want to dictate that all hair is yellow, for instance), select none in the Add properties drop-down(s) in the Add Related Concepts box.

HOW DOES IT KNOW THAT WEIGHT HAS … WELL, WEIGHT?

Again, it doesn't initially; it has to be taught. In this case, since weight is an ur concept, you'd enter weight as a new ur, then give it members (such as heavy, light, etc.). This is covered in more detail in the Ur Concepts section.

CHANGING AND DELETING PROPERTIES

If you already have your related concept, but you want to change the value of one of its properties, you can click the properties dropdown (located after the words "which is/are" onscreen), select the property you wish to change, and clicking the Chg Prop button. You'll get a popup similar to the one below:

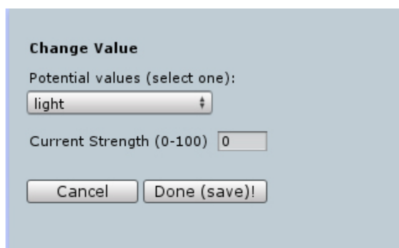

FIGURE 4

You will be able to choose from amongst the values in the dropdown, and you can change the strength of the property in the character's memory. Note that lower property strengths indicate less sureness on the character's part that this is the correct value; a property strength of 25 for "red" as the color of the Healer's hair would

indicate that the Fighter isn't very sure about her hair color, and that his mind might be changed easily if presented with conflicting information.

You can also delete a property by selecting the Del Prop button in the main window.

This memory is part of the concept …

Sometimes you'll have memories that are part of larger concepts; e.g., Healer is part of a larger concept called person. These larger concepts are always ur concepts.

You can change an ur concept by pressing the Change Ur button under the concept dropdown. The window will give you a dropdown list of all ur categories that could be this higher concept.

To delete the higher concept from the particular memory, press the Del Ur button.

It's all a matter of context

The next dropdown contains a list of the contexts in which this memory was learned, and which can be used to recall the memory if it is obfuscated ("forgotten"). Context items should be other memories, but can be anything (and hence have their own list that doesn't necessarily contain the memories of this character). To add context, press the Add Ctext button. You can choose an existing context, or add a new one. You will then be given the opportunity to give the context a strength; the higher the strength, the more helpful it will be in retrieving memories associated with it.

To delete a context item, press the Del Ctext button.

Obfuscation and permanence

The next row begins with two toggle buttons. Ticknig the first will mark the memory as already obfuscated; that is, the character has this memory, but it is forgotten and inaccessible unless given the right context or emotional cue.

Ticking the second toggle marks the memory as permanent; it will never lose strength and become distorted or obfuscated due to memory maintenance (although it can still begin as obfuscated if you have so marked it).

Stories

As a convenience, all the stories a character knows are listed here. They are editable on the Stories screen, discussed later. See Using Only Stories, above, for a discussion of stories.

On to the Ur

The ur list section displays all the character's current ur concepts and gives you the ability to add or edit them. Again, ur concepts are overarching concepts (e.g., color) that include subconcepts (e.g., blue, green, red, etc.).

The first box in the row displays the character's current ur concepts. Adding an ur is much like adding any other memory: Click the Add Ur button, and then type in the name of a new ur concept. There is no strength for ur concepts; they are assumed to be strongly embedded in memory. As with regular memories, you can add multiple urs at once by separating them with a comma (no space), as, for example: color,shape,length

To delete an ur concept, select an ur from the dropdown, and click the Del Ur button.

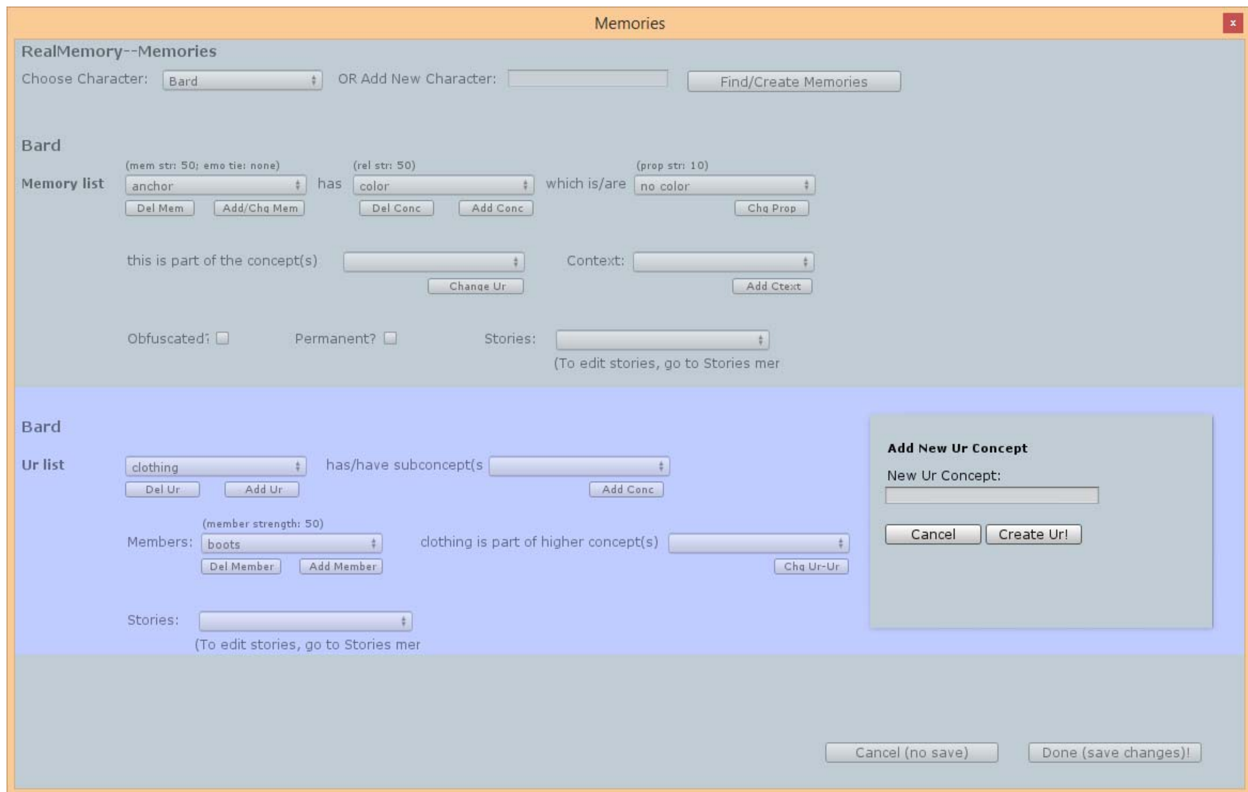

FIGURE 5

The next dropdown in that row is for the subconcepts (related concepts) of the ur concept. For instance, the ur concept person has the subconcepts hair, eyes, skin, height, weight, etc.; that is, every person has these basic things (or most people do). To add such a related concept, press the Add Conc button, and in the popup window, select the concept to add. There are no properties of strengths to add, as this is a generalization; sort of a prototype.

To delete a subconcept, press the Del Conc button.

MEMBERS

The next row begins with the dropdown for members of the ur concept; these are the items that show up as properties in regular memories. Members of color would be red, blue, green, etc. To add a member, select the Add Member button; the popup window will give you a list of all the memories in the character's mind, and you can choose any of these to be a member of the ur concept. Sadly, at this time you can choose only one new member at a time; the Unity editor interface doesn't include a multiple selection dropdown list.

To delete a member, select the Del Member button underneath the member dropdown in the main window.

HIGHER AND HIGHER …

You can also make this ur concept part of an even higher ur concept, for instance making person a part of living beings. Click Add Ur-Ur to get a popup with a list of potential urs; click the Del Ur-Ur button to delete the Ur-Ur already here.

Stories

Finally, the character can have stories about ur concepts, too. A list is provided here, but the Stories are editable on the Stories screen.

Finally … SAVE!

Finally, don't forget to save all your work! All changes are unsaved until you click Done (Save)! At the bottom right of the Memories window. Clicking Cancel gets rid of everything you've done during this session.

Telling Tales: The Stories Window

For more on what stories are and how to use them in-game, see Using Only Stories. To create stories, go to the Stories submenu of RealMemory.

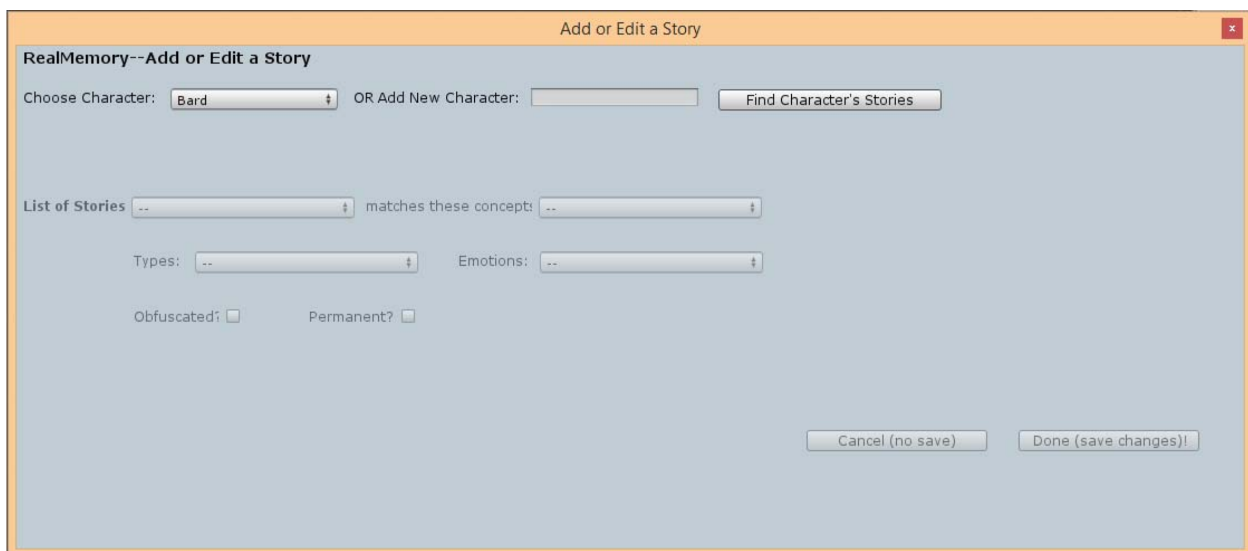

FIGURE 6

Choose character, or add new character

As in the other windows, the top row allows you to choose from an existing character from a drop-down list, or to type in the name of a new character in the text box. Then click the Find/Create Memories button to bring up the character's current memories.

Note that if there is any text in the text field, RealMemory will assume that you want to create a new character with that name. (And if you type in the name of an existing character, it will retrieve that instead.)

List of Stories

The first dropdown displays all the stories a character knows. To add a story, click Add Story and type your story into the text field, then click Done. To delete a story, click the Del Story button.

Each story has several potential elements. The next dropdown asks for any concepts the story might match. These are something like context clues in the character's memories, except that story concepts are used to retrieve stories about a particular memory. Say the Fighter remembers that the Healer loves to go down to the pub and sing the occasional song, or that she used to have green hair and sang in a punk band. These stories would have

matching memory concepts like “Healer”,”green”,”hair”,”pub”,”song”, etc. Even if using only stories as your memory system, these concepts will be created as memories (just with no links between them).

To add a concept, click Add Conc and either type in a new concept or use an existing memory as a concept. To delete a story concept, select the concept in the dropdown and click Del Conc.

Story Types

Types allow you to give the story a context unavailable to other memories—for instance, you can give the story a “Past” type to indicate that it was true in the past, but not now (such as The Healer’s hair was green), or as “OtherOpinion” to indicate that this story is someone else’s opinion (e.g., “The Thief thinks the Healer’s singing voice is great”), or anything you’d like. To add a story type, click Add Type and either select one from the list or create a new one, then click Done. To delete a type from this story, click Del Type.

This story has feeling

You can also create emotional ties to stories, just as you can with regular memories. These emotional ties can act as context clues to remind a character of a story, or remembering the story can make the character feel that emotion. To add an emotion, click Add Emotion and either select one from the list or add your own, then click Done. To delete an emotion from this story, click Del Emotion.

I can’t quite remember …

Stories, like other memories, can become obfuscated or be made permanent (thus avoiding obfuscation). You can set that value here by ticking the appropriate box.

Save! Or don’t

Remember to click Done (Save)! in the lower right of the Stories window to save your work, as nothing is saved until you do this. Or of course you can click Cancel to get rid of the changes you’ve made in this session.

Create Memory from Overarching Concept: The Create from Ur Window

In this window you can create, quickly and easily, an entire specific memory based on an ur category. For example, say your Fighter knows all about the ur category “people” (that is, he knows people have hair, eyes, etc.). But now you’re faced with having to enter all the specific people he knows, and then enter that each of them has eyes, and hair … that could be very tedious! But you don’t have to do it that way; instead, you can build all the people he knows from the ur concept using this window. (Note that it won’t give you the specifics (such as the Healer’s hair is red), but you’ll have everything else set up, so you can go into the main memories window and enter the specific hair color, if the Fighter knows it.)

To use this, first you need to select or enter the character whose memories you’re altering, as in the other windows. Either choose a character from the popup list, or enter a new one in the text box, and click Find/Create Character.

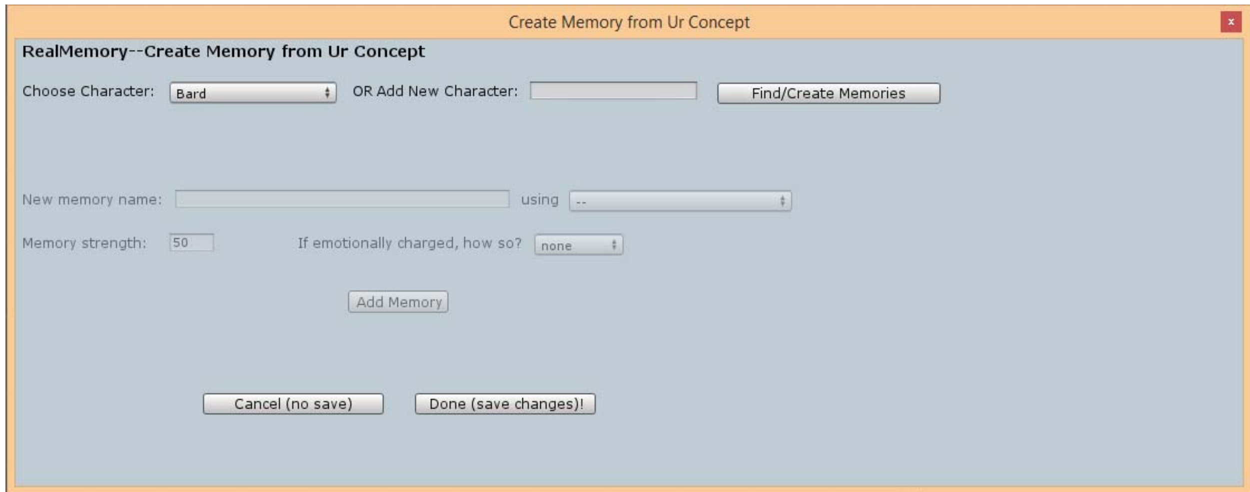

FIGURE 7

You will then be able to type in the new memory name (say, Healer), then select an ur category to use to build the memory (in this case "person"). You can also change the memory strength and whether or not there is an emotional tie to this new memory. Then click the Add Memory button.

You can add as many of these new memories as you like. You can enter them one at a time (pressing Add Memory after each) or enter a list of memories, separated by commas (but no spaces between entries), like this:

Healer,Bard,Thief

Pressing Add Memory will then add all the characters to memory, assuming they don't already exist.

Remember that you have to click Done in the lower right to save, just as in the other RealMemory windows, or you will lose your changes. Cancel, of course, cancels all changes made during this session.

Duplication: The Copy Memories Window

Say we have the Fighter set up, but we have other characters who should have the same (or nearly the same) memories. Rather than do it all again for these other characters, you can copy all memories from one character to another.

In the Copy Memories window, you first choose whether to copy all memories and stories, memories only, or stories only. This can be helpful if characters share stories but not memories, or vice versa.

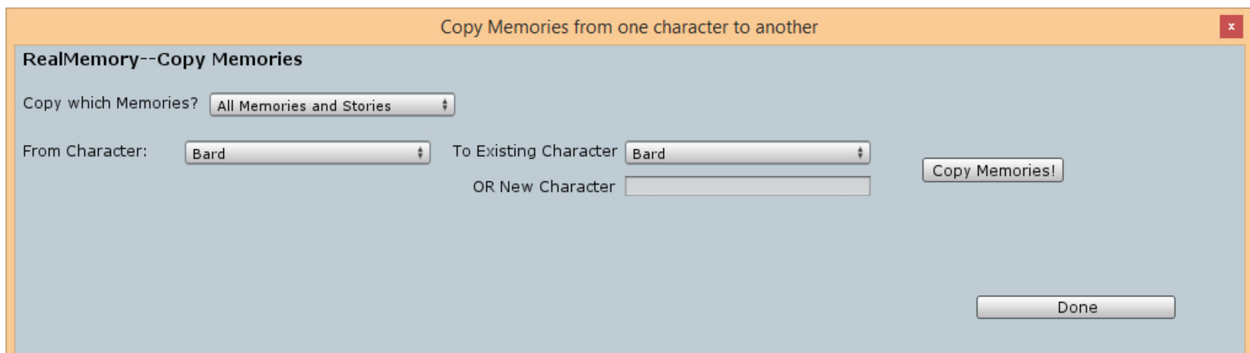

FIGURE 8

In the next box you choose which character you're copying from. This of course is an existing character (you can't copy from one that does not exist). Then you can choose who to copy to. You can choose either an existing character or a brand new character. Note that if any text is in the new character name box, copying will default to that new character.

If you copy to an existing character, any memories of the same name will be overwritten by the memories being copied (exception: if copying stories only, any story concepts that match existing memories in the "copied to" character won't overwrite those memories, as the story concepts need only exist). For instance, if copying from the Fighter to the Healer, if both have a memory of anchors, the Fighter's memory will overwrite that of the Healer, and she will think of anchors in exactly the same way the Fighter does. If, however, the Healer knows something the Fighter does not, that memory will remain and not be overwritten (she knows about medicine, but he does not; she will still know about medicine after the copying process). If copying stories only, and both the Fighter and the Healer know about anchors but think differently about them, any of the Fighter's anchor stories will now exist in the Healer's memory, but her thoughts about anchors themselves will still be the same as they used to be.

Clicking Copy Memories copies the memories and/or stories. In this case, the Done button merely closes the window.

Register RealMemory!

There are two other menu items—About QTG, which tells you about us (and displays the version number), and Updates and Registration, which allows you to register your copy of RealMemory with us.

Support and Credits

Support

We're here to help! Contact us at support@quantumtigergames.com with any issues.
We'll also post a FAQ on our website (once we have enough questions to have a FAQ!).

Credits

Design, coding: Jeffrey Georgeson
Raina character created by Mark Foley. Copyright © and TM 2012 Quantum Tiger Games, LLC