Version 1.7

WHAT IS LOVE/HATE?

Love/Hate is a relationship and personality simulator for Unity. It models characters' feelings about each other using emotional states and value-based judgment of deeds.

With Love/Hate, your characters will…

- **…have personalities and emotional states**.
  Love/Hate uses powerful, customizable personality and emotional state models.
- **…maintain relationships**.
  Love/Hate tracks how different characters and groups (factions) feel about each other, allowing you to define a dynamic web of social ties among your characters.
- **…judge and remember deeds**.
  Characters witness deeds committed by others, judge them according to their own personality and relationships, and remember them.
- **…gossip with other characters**.
  Characters share memories realistically, allowing news of actions to spread organically.

Love/Hate doesn't replace reason-based AI such as finite state machines and behavior trees, but instead works alongside them to provide interesting, character-based motivation.

Love/Hate is the product of several years of development in realtime simulation of emotions and value-based appraisal, incorporating the latest academic and industry research. Despite its sophistication, Love/Hate is very lightweight and CPU efficient at runtime. It includes complete, fully documented source code.

Love/Hate also supports Hutong Games' *PlayMaker*, Opsive's *Behavior Designer*, Icebox Studio's *Adventure Creator*, Gaming Is Love's *Makinom* and *ORK Framework*, Gavalakis Vaggelis' *Node Canvas*, CallumP's *TradeSys*, and Pixel Crushers' *Dialogue System for Unity*. (These products must be purchased separately.)

Table of Contents

INSTALLATION

Love/Hate imports into this folder structure:

Pixel Crushers:
- **Common**: *Shared assets used by Pixel Crushers products.*
- **LoveHate**:
  - **Example**: *Examples of Love/Hate's features; you can safely delete this.*
  - **Scripts**: *Complete source code.*
  - **Templates**: *Starter templates for faction databases and deed templates.*
  - **Third Party Support**: *Packages to help integrate with third-party products.*

You can find the package version in Pixel Crushers/_README.txt and version release notes in Pixel Crushers/_RELEASE_NOTES.txt.

EXAMPLE SCENE

The example scene demonstrates how the player's actions affect NPCs based on their personalities and their relationships to the target of the action. You play a visitor to a tiny kingdom with a bandit problem. The scene contains four NPCs that you can interact with. When you bump into an NPC, you can see her current faction information and click buttons to perform actions such as Flatter and Insult. As you perform actions, watch the faction information window and the NPCs' animations, which are driven by Love/Hate.

The subtle halo around each NPC is her greeting range. When other characters enter it, she will greet them according to how she feels about them. If she likes the characters, she'll tell them about any of your actions that she's witnessed. This may change how those characters feel about you.

The third party support packages come with their own example scenes. You can read more about these examples in their respective sections of the manual.

SETUP OVERVIEW

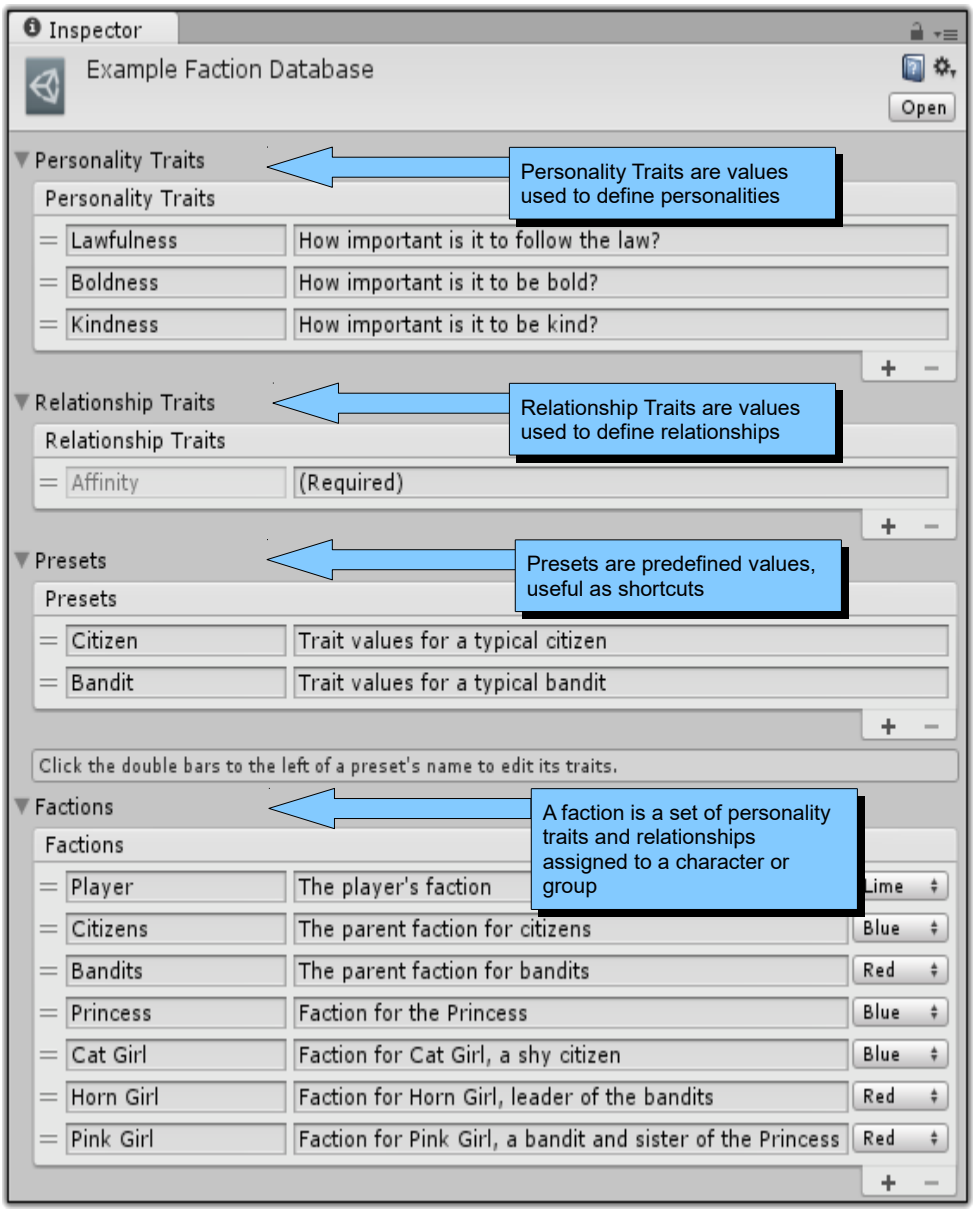
To set up Love/Hate, you'll follow these steps:

1. Create a **faction database**.
2. Configure **faction members** (characters).
3. Inform Love/Hate when **deeds** (game actions) occur.

The sections below describe these steps in detail.

FACTION DATABASE

A **faction database** is an asset that contains the following user-definable information:



To create your faction database, you can copy a template or create a new, empty faction database.

Using a Template Faction Database

The **Templates** folder contains starter databases that use various psychological models such as the “OCEAN” five-factor model commonly used in behavioral psychology. To use a template, select it in the Projects view and press **Ctrl-D** (⌘**-D**) to duplicate it.

Using an Empty Faction Database

To create a new, empty faction database, select menu item **Assets > Create > Love/Hate > Faction Database**.

Traits

A **trait** is a value in the range [-100,+100], where +100 means “strongly agrees” and -100 means “strongly disagrees.” Traits apply to personalities and deeds. Love/Hate lets you define the traits you want to use in your project. You can make them as streamlined or complex as you want. The Templates folder contains several starter databases with predefined traits for psychological models such as the “OCEAN” five-factor model commonly used in behavioral psychology.

> **Example**: In the example scene, the faction database defines a trait named *Lawfulness*. The Princess, a ruler devoted to upholding the law, has a high *Lawfulness* value. Horn Girl, a Robin Hood-style bandit, has a negative *Lawfulness* value.

Traits are what characters *value*, not necessarily what they do themselves. A character might not be confident herself, but she might be attracted to confidence in others.

To define a new trait, click the “**+**” button in the lower right of the Traits list.

Presets

A **preset** is a predefined set of traits, useful as a shortcut. You can use presets to assign default values to factions.

> **Example**: In the example scene, the *Bandit* preset defines trait values for a typical bandit. This preset was applied to Horn Girl and Pink Girl. Then they were customized further to make each character unique. Later in the manual, you'll see that traits will also apply to deeds and other entities such as locations. You can define and apply presets to these, too.

To define a new preset, click the “**+**” button in the lower right of the Presets list. Enter a name and optional description. Then click the horizontal bars to the left of the preset name to edit its values.

If you want to organize presets in groups, you can use forward slash characters in the preset name.

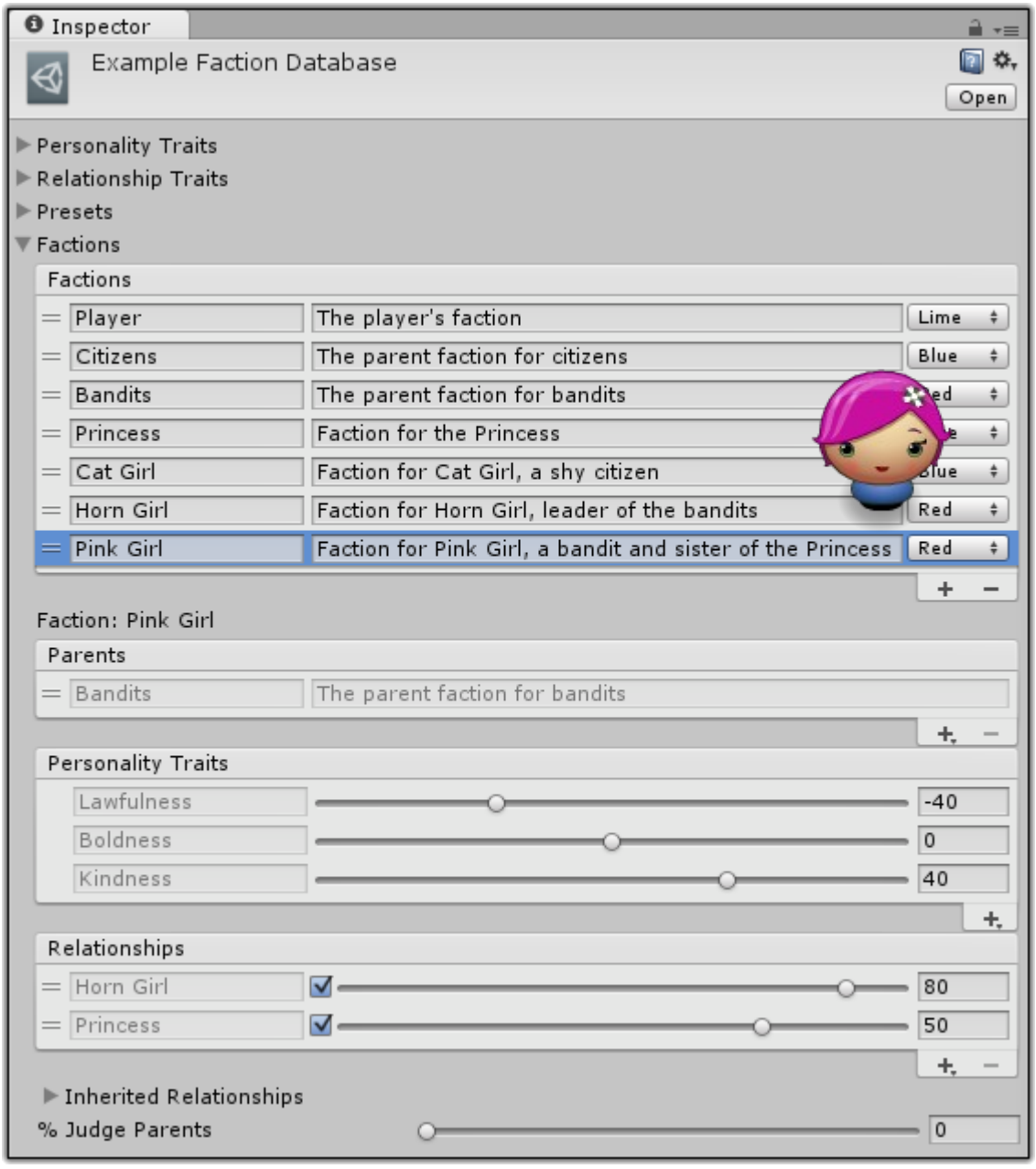
> **Example**: Say you have several presets for monsters. You can name them *Monsters/Orc*, *Monsters/Goblin*, *Monsters/Kobold*, and so on. In faction menus, they will all be grouped in a submenu *Monsters*.

Note: If you change a preset’s values, characters or deeds to which you’ve previously applied the preset will *not* be automatically updated with the new preset values.

Factions

A **faction** is a social circle. It defines a set of personality traits and relationships. Usually a character will have its own personal faction, although interchangeable generic NPCs, such as generic goblins, might share a common faction instead. A character who belongs to a faction is called a **faction member**.

To define a new faction, click the "**+**" button to the lower right of the Factions list. Enter a name, optional description, and color for this faction's gizmo in the Scene view. Then click the horizontal bars to the left of the faction's name to edit it. If you want to organize factions in groups, you can use forward slashes in their names, such as *Titans/Cronos* and *Titans/Themis*.



Parents

A faction can belong to any number of parent factions. To add a parent, click the "**+**" button in the lower right of the Parents list. You only need to specify direct parents, as those parent factions may have their own parents (this faction's grandparents) and so on. Above, Pink Girl has Bandits as a parent. She inherits any relationships that Bandits have toward other factions.

Personality Traits

Use the sliders or input fields to set the faction's personality trait values. To apply a preset, click the "**+**" button in the lower right of the Personality Traits list. If you select "*(Average from parents)*", it will average the current values of any assigned parents. If you select "*(Sum from parents)*", it will add the parents' values. (Factions don't inherit these values from their parents by default.) In the screenshot on the previous page, Pink Girl's Lawfulness is -40, indicating that she has disregard for the law. Her Kindness is 40, indicating that she values kindness.

The **Trait Inheritance Type** dropdown lets you select *Average* or *Sum*. This specifies how personality traits are inherited from parents when calling FactionDatabase.InheritPersonalityTraits() in your scripts at runtime, or using equivalent visual scripting methods.

Relationships

Factions have relationships to other factions. Each relationship specifies a set of relationship traits in the range [-100, +100]. There will always be an **Affinity** trait, where +100 indicates absolute love and -100 indicates absolute hatred. In the screenshot on the previous page, Pink Girl has a strong +80 affinity to her idol, Horn Girl, and a moderately strong +50 affinity to her sister, the Princess.

You can choose to add more relationship traits, such as Rivalry or Obligation. You might use Obligation in an honor-based environment such as a traditional samurai tale. Rivalry makes a good counterpoint to Affinity in romance stories.

> **Example**: In a sci-fi RPG, Drake, the xeno-archaeologist player character, is forced to travel with Ariana, a tough artifact smuggler. Ariana starts with low *Affinity* to Drake and high *Rivalry* – she wants to steal the artifacts; Drake wants to preserve them. As they travel the galaxy, Ariana begins to appreciate Drake's views and abilities. Her *Affinity* increases and her *Rivalry* decreases until, at some point, romance can bloom. If her *Rivalry* remains too high, however, it might override her growing affection and prevent romantic options.

Note that relationships are one-way.

> **Example**: A third character has a secret crush on Drake. She has a high *Affinity* to him, but Drake doesn't even know that she exists, so he has no *Affinity* to her.

To add a relationship, click the "**+**" button in the lower right of the Relationships list.

Relationships can be inheritable (the default) or non-inheritable. To mark a relationship non-inheritable, untick the checkbox next to the subject's name.

To view relationships that are inherited from parents and ancestors, expand the **Inherited Relationships** foldout. This will display a new section containing all inherited relationships and which ancestors they're inherited from.

The **Relationship Inheritance Type** dropdown lets you select *Average* or *Sum*. This specifies how relationships are inherited from parents.

The **% Judge Parents** value, if non-zero, specifies how much to affect relationships to the subject's parents when updating a relationship to a subject at runtime.

FACTION MEMBERS AND FACTION MANAGER

Factions vs. Faction Members

Love/Hate characters are **faction members**.

It's important to understand the distinction between **factions** and **faction members**.

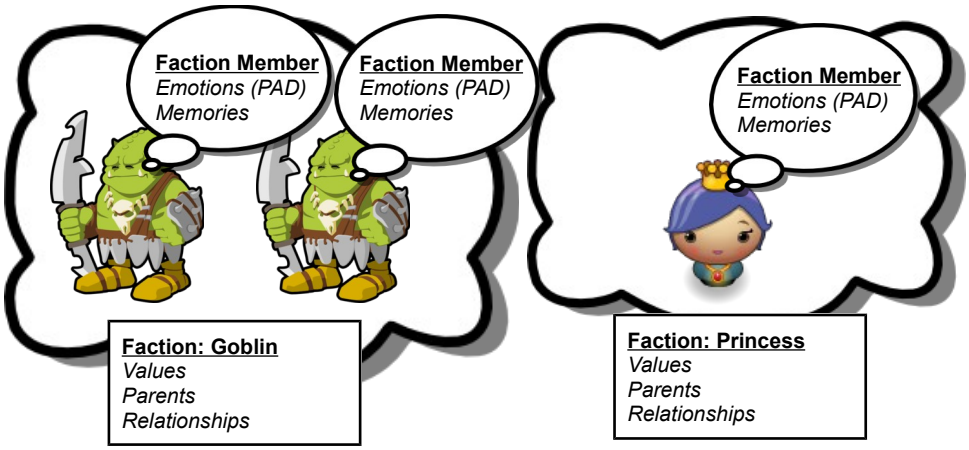
A **faction** is defined in a **faction database**. Its properties are relatively abstract. More than one faction member can belong to a faction.

A **faction member** exists on a GameObject in a scene. Its properties deal with the specific experiences of the GameObject – what it sees (witnessing deeds) and how it feels (temperament).

> The “hive mind” is a classic sci-fi trope, epitomized by the Borg in *Star Trek* and the Zerg in *StarCraft*. You could think of a *faction* as the hive mind. Each *faction member* is an individual unit with its own eyes, ears, and mouth.

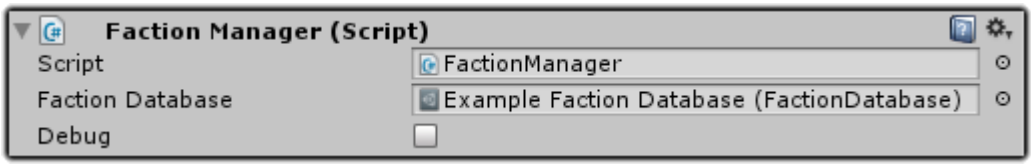
For unique characters, there will be a one-to-one relationship between a faction member and her unique faction. For generic characters, multiple faction members may share the same faction.

Faction	Faction Member
Abstract (Faction Database) Values, Parents, Relationships	Instanced (GameObject) Emotions (PAD), Memories



Faction Manager

The **faction manager** helps faction members talk with each other in the scene. If your scene has any faction members, it should also have a faction manager.

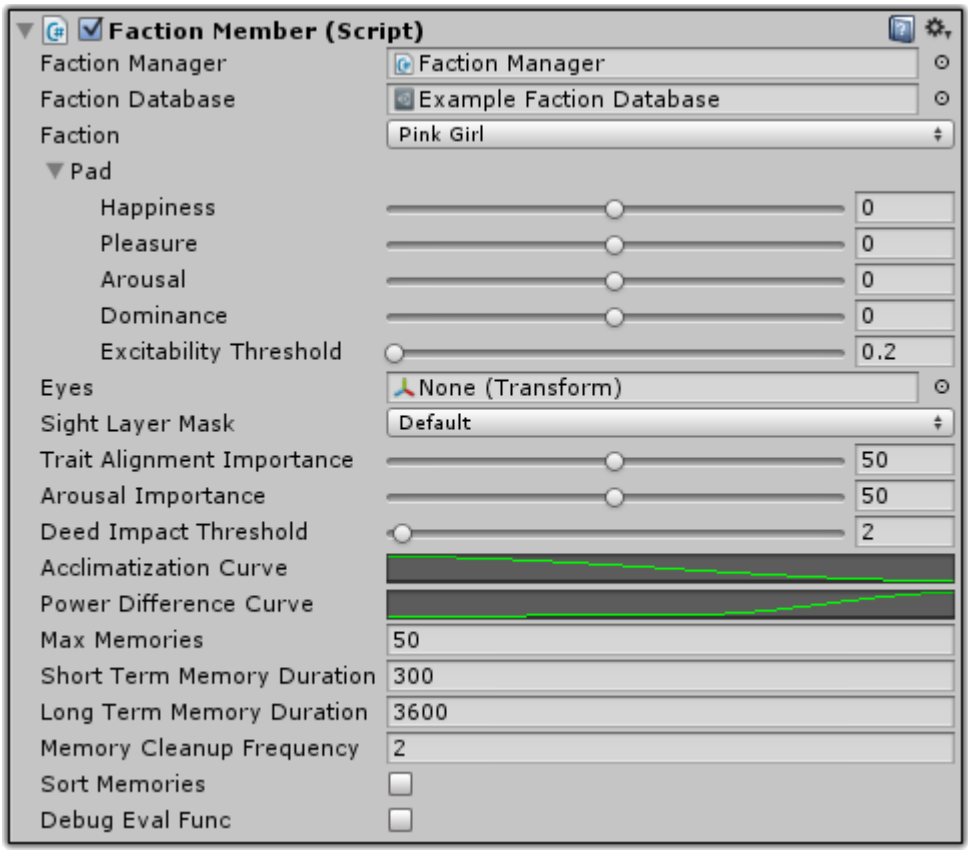


To set it up a faction manager:

1. Select menu item **GameObject > Love/Hate > Faction Manager**.
2. Assign your faction database.
3. If you want to log Love/Hate activity to the console, tick **Debug**.

Faction Member

A **faction member** is an NPC (a GameObject in your scene) that belongs to a faction.



To configure an NPC as a faction member, select it, and then select menu item **Component > Love/Hate > Faction Member**.

If your scene already has a faction manager, the Faction Member script will find its faction database. Otherwise assign your faction database so you can select the faction from the dropdown menu.

Faction members share these values with all members of the faction:

Shared Faction Values

- Parents
- Values
- Relationships

In addition, each faction member stores its own data in the fields shown in the image above. Those fields are described below.

PAD (Pleasure-Arousal-Dominance)

PAD is an implementation of the Pleasure-Arousal-Dominance emotional model. (See https://en.wikipedia.org/wiki/PAD_emotional_state_model for more information about this model).

The values are in the range [-100,+100].

PAD Values

- **Pleasure** (P): How happy or sad the character currently is.
- **Arousal** (A): How worked up and excited the character currently is.
- **Dominance** (D): How dominant or submissive the character currently feels.
- **Happiness**: The sum total of all pleasure values (positive and negative) over the character's lifetime.
- **Excitability Threshold**: The threshold above which the PAD values affect temperament. Temperament is an emotional state, such as *Disdainful* or *Exuberant*, based on the character's PAD values. A high excitability threshold means the character is less easily moved into an emotional state.

Temperament	P	A	D
Exuberant	+	+	+
Bored	-	-	-
Dependent	+	+	-
Disdainful	-	-	+
Relaxed	+	-	+
Anxious	-	+	-
Docile	+	-	-
Hostile	-	+	+
Neutral	0	0	0

If you want a character's PAD values to gradually return to target values over time, add an **Equalize PAD** component, which is described later in the manual.

If you want to define a more complex emotional model than Temperament, you can create an **Emotion Model** and add an **Emotional State** component to your faction member. This is described in its own section further down.

Vision

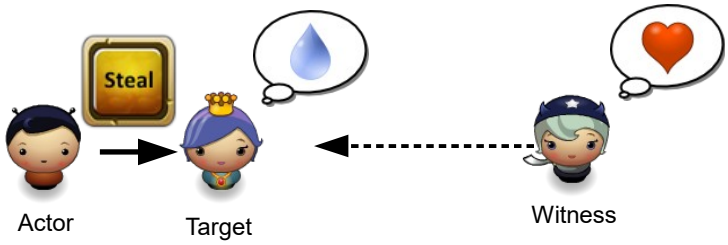
Eyes

Some deeds can only be witnessed if the NPC can see them being committed. To determine visibility, the NPC runs a raycast from the Eyes transform to the actor. If Eyes is unassigned, it will use the NPC's transform, which is usually at its feet.

Sight Layer Mask

The sight layer mask specifies which layers to check when running the visibility raycast.

Memory



Characters remember **rumors**, which are subjective memories of deeds that they've witnessed or heard about. Short term memory affects the character's PAD state. Long term memory is used to gossip with other characters.

> **Example**: If you steal from the Princess, she'll remember a negative rumor about you. Her affinity to you will decrease. It will decrease her Pleasure and increase her Arousal. If you're more powerful than she is, her Dominance will decrease. When she encounters a friend, she'll share this rumor. The friend will remember her own rumor about your deed.
>
> If the bandit Horn Girl sees you steal from the Princess, or hears about it from a friend, she'll also remember a rumor about you. But, since she doesn't like the Princess, this rumor will improve her affinity to you and increase her Pleasure.

The values described below influence how the character subjectively evaluates the impact of a deed. They define how much importance the character places on different aspects of deeds.

Trait Alignment Importance

This value affects how much trait alignment affects the impact of a deed. When a deed's trait values exactly match the character's trait values, alignment is 100%. When the deed's trait values are in complete opposition with the character's trait values, alignment is -100%. If you set **Trait Alignment Importance** to 100, this will fully double the impact of deeds that completely match or oppose the character's traits. If you set it to 0, trait alignment will not affect the impact at all.

Deed Impact Threshold

All witnessed deeds will affect the character's PAD, but only important deeds are worth remembering. The character will only remember rumors whose perceived impact is higher than this threshold. The perceived impact of a deed is determined by how well the deed aligns with the character's values and the character's affinity to the target.

Acclimatization Curve

When characters witness repeated deeds on the same target, such as repeated flattery, the deed's impact will diminish. The acclimatization curve specifies how quickly the impact falls off. With the default value, the character will stop caring any further about the deed after 20 repetitions. However, once the deeds fall out of memory the curve is reset.

Power Difference Curve

This curve influences how the character's dominance value is affected. When a character learns about a deed that helps a friend (include the character itself), it gains dominance. When a character learns about a deed that injures a friend, it loses dominance. The Power Difference Curve helps negate dominance loss and increases dominance gain. The curve values are on a scale of 0 to 1, where 1 means an additional 100% of the dominance change is gained.

> **Example**: The actor casts a Fear spell, which normally decreases dominance by 10. The character is power level 6. The actor is power level 1. The power difference is 6 – 1 = **5**. The character's Power Difference Curve at 5 is 0.8 (80%). 80% of 10 is **8**. Since the character is more powerful, instead of taking a -10 hit to dominance, it only takes a hit of -10 + 8 = **-2** to dominance.

Max Memories

To conserve memory, characters limit the number of things they'll remember. When memory is full, old rumors will be forgotten to make room for new. If you tick **Sort Memories**, lower-impact rumors will be forgotten first.

Short Term Memory Duration

This is the maximum amount of time that a rumor will stay in short term memory. The actual duration of each rumor will be scaled down based on the faction member's perceived importance of the rumor, which is equivalent to how much it affects the faction member's affinity to the actor. If it increases the affinity by +50 (half of the maximum +100), the rumor will stay in memory for only half the duration specified by this field.

While a rumor is in short term memory, it affects the character's PAD value.

Long Term Memory Duration

This is the maximum amount of time that a rumor will stay in long term memory, which is scaled down based on the faction member's perceived importance of the rumor.

While a rumor is in long term memory, the character can share it with friends it encounters.

Memory Cleanup Frequency

The character will check for expired memories on the frequency (in seconds) specified by this field.

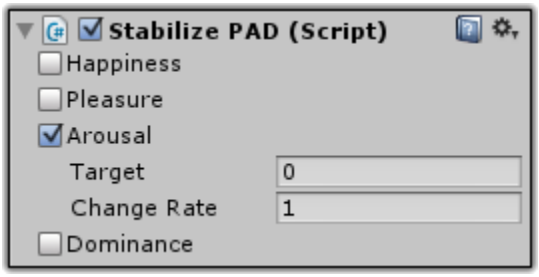
Sort Memories

If ticked, memories will be sorted in increasing impact. This makes it easier for your scripts to find the most important rumor in a character's memory. There is a slight extra overhead required to insert the memory in sorted order, but in all but the most extreme cases the overhead is negligible.

Debug Eval Func

If ticked, the faction member will log its “though process” to the console when it evaluates a rumor. You may find this information useful to help tweak your faction database settings.

Stabilize PAD

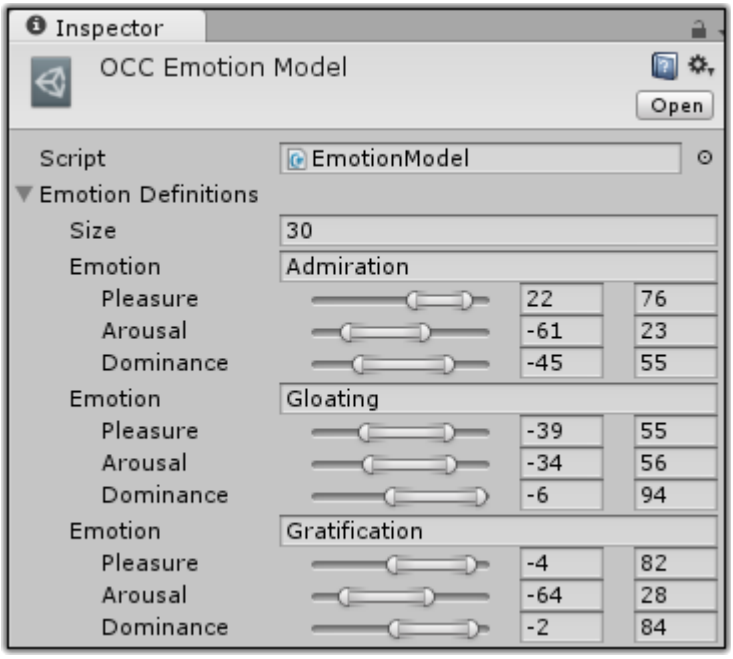


To make a faction member's PAD values gradually return to target values over time, add a **Stabilize PAD** component by selecting menu item **Component > Love/Hate > Stabilize PAD**. Tick the values you want to affect, and then enter the target value and rate of change per second. In the screenshot above, arousal will stabilize toward zero at the rate of 1 per second.

In the screenshot above, the Stabilize PAD component will gradually bring the faction member's Arousal to 0 at a rate of 1 unit per second.

Emotion Model

If you want to define a more complex emotion model than the PAD's Temperament, you can create an Emotion Model and add an Emotional State component to your faction member(s). To create an emotion model, select menu item **Assets > Create > Love/Hate > Emotion Model**.



You can define any number of emotions. In the screenshot above, if a faction member's Pleasure is in the range [22,76], Arousal is [-61,23], and Dominance is [-45,55], the faction member's emotion will be *Admiration*. The Templates folder contains an emotion model that maps the 22 OCC emotions defined by behavioral psychologists.

To apply an emotion model to a faction member, add an **Emotional State** component and assign the emotion model to the **Template** field. You can customize the model for each faction member. For example, if a faction member is particularly fearful, you can increase the ranges for the *Fear* emotion.

At runtime, the inspector will show the faction member's current emotional state.

TRAITS ON OTHER OBJECTS

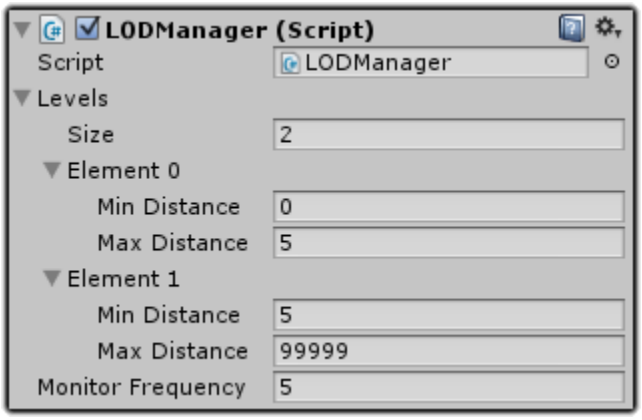
You can add a **Traits** component to other non-faction member GameObjects to associate traits with them. To add a traits component, select the GameObject, and then select menu item **Component > Love/Hate > Traits**.

Love/Hate doesn't do anything itself with GameObjects that have Traits components. Instead, your own scripts or PlayMaker actions can examine the GameObject's traits to help make decisions. You could add traits to locations, items, even other creatures that aren't faction members.

> **Example**: In folklore, vampires can't enter churches or cross streams. A vampire game's faction database defines a personality trait named *Purity*. In the scene, the church and stream GameObjects have Traits components with high *Purity* values. When the vampire AI script decides where to go, it can try to avoid GameObjects with a high *Purity* value.

You can use Traits in conjunction with an **Aura Trigger** (described further down in the manual) to affect characters that enter a trigger collider. The effect is based on how the character's traits align with the Aura Trigger's traits.

LOD MANAGER



The Pixel Crushers common scripts include LODManager, which is a basic level-of-detail manager.

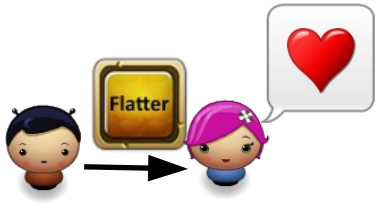
To use LODManager, add the script to your character and define the level-of-detail distances. Set Monitor Frequency to the frequency (in seconds) at which the LODManager should check distance from the player to update the current level of detail.

You can use LODManager to automatically adjust a faction member's memory cleanup frequency based on the character's distance from the player. When the player is close, the character will clean up memory according to the value of **Memory Cleanup Frequency**. When the player is far, it will reduce the frequency to conserve CPU.

To use level-of-detail in your own scripts, simply implement the `OnLOD(int level)` method:

```
public void OnLOD(int level)
{
    // Change behavior based on the value of level,
    // where level 0 is the closest to the player.
}
```

DEEDS



A **deed** is a gameplay action that NPCs can witness. It consists of:

Deed

- **Tag**: An identifying string such as “Attack” or “Flatter” used to categorize deeds
- **Actor**: The character performing the deed, usually the player
- **Target**: The character the deed is performed on
- **Impact**: A value in the range [-100,+100] indicating how harmful or beneficial the deed is
- **Aggression**: A value [-100,+100] indicating how aggressive or submissive the deed is
- **Traits**: Trait values that define the nature of the deed

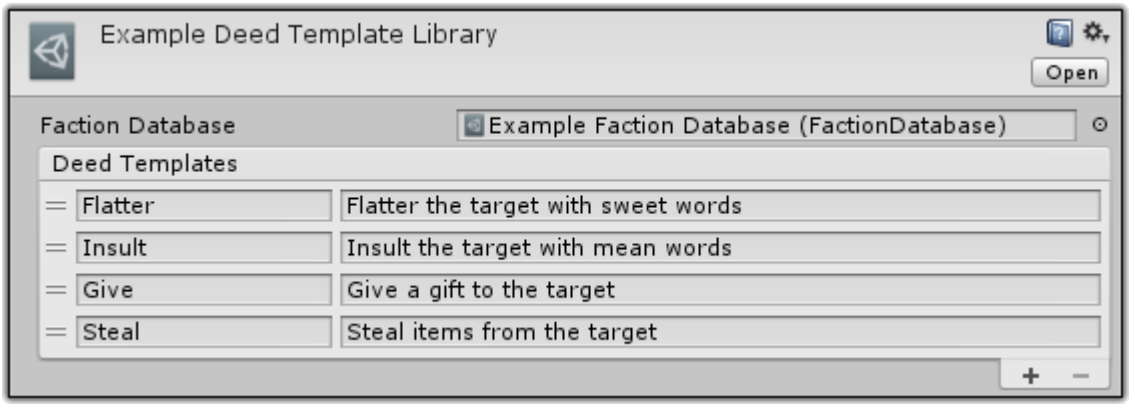
The most direct way to report a deed is to call `FactionManager.CommitDeed()` in a script. This is described in detail in the Scripting section of the manual.

An easier way to report deeds is to use a **Deed Template Library** and a **Deed Reporter**.

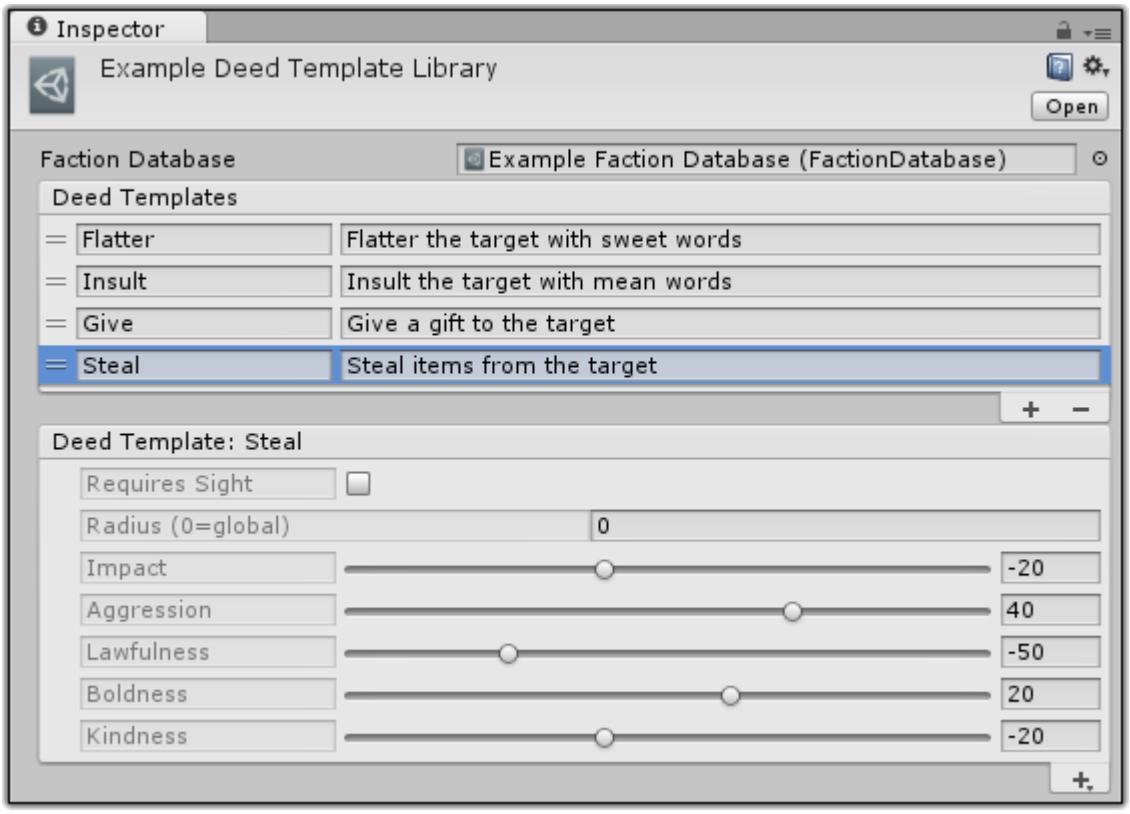
Deed Template Library

A **deed template** is an abstract definition of a deed. It defines general attributes without specifying an actor or target. Deed templates are contained in a **deed template library**.

To create a deed template library, select menu item **Assets > Create > Love/Hate > Deed Template Library**.



To add a deed template, click the “**+**” button in the lower right of the Deed Templates list. Enter the deed template's tag and optional description. Then click the double bars to the left of the deed's tag to edit its attributes.



Requires Sight

Some deeds can only be witnessed if the character can see them being committed. Tick this if characters should perform a visibility check to witness the deed.

Radius

This is the reporting radius of the deed. If you want all characters to be able to witness the deed, set Radius to 0.

Impact

Use the slider or edit field to set the deed's impact, where -100 is the worst possible thing that could happen to the target and +100 is the absolute best. In the example above, flattery is a minor deed, so its impact is only +5.

Aggression

Use the slider or edit the field to set the deed's aggression, where -100 is the most submissive and +100 is the most aggressive. Aggression affects characters' dominance. For example, witnessing aggressive deeds against friends will make characters feel submissive.

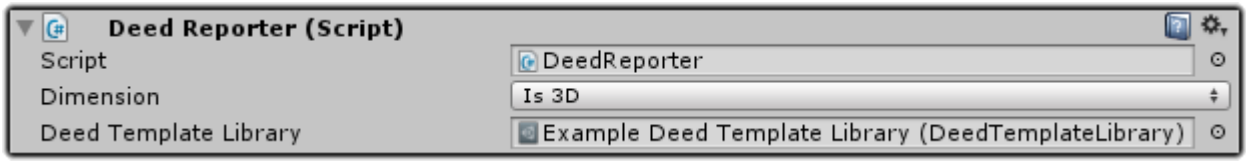
Traits

The remaining fields in the template are the trait values. Characters will compare the deed's trait values to their own trait values. The degree of alignment will modify the impact the deed has on them.

> **Example**: The *Flatter* deed has a *Kindness* value of 70. If a character has a negative *Kindness* value, indicating that she disdains kindness, the deed will not align with her values, and the deed will have less impact on her.

Deed Reporter

A **deed reporter** uses deed templates to report deeds committed by an actor to a target.



The **DeedReporter** script goes on the character (usually the player) that will commit deeds. To add a deed reporter, select the character, and then select menu item **Component > Love/Hate > Deed Reporter**. Then assign a deed template library.

Deed reporters work in 3D and 2D scenes. Specify the type of scene in the **Dimension** dropdown.

Tying Deeds to Gameplay

When the character commits a deed, call the `ReportDeed()` method. Provide the deed template tag and the target:

```
GetComponent<DeedReporter>().ReportDeed("Flatter", pinkGirl);
```

You can also use equivalent PlayMaker actions, Dialogue System functions, etc., to implicitly call `ReportDeed()`. These are described in their own sections later in the manual.

For a more detailed example, say your player has a Combat script, and you want to report whenever the player attacks another character in the Attack() method. Add these lines to your script:

```
void Attack(GameObject target)
{
    animator.SetTrigger("Attack");
    target.SendMessage("TakeDamage", weaponDamage);
    var targetFactionMember = target.GetComponent<FactionMember>();
    if (targetFactionMember != null)
    {
        GetComponent<DeedReporter>.ReportDeed("Attack", targetFactionMember);
    }
}
```

(Ideally you'll cache DeedReporter in Start() rather that finding it every time the player attacks.)

Deed Evaluation

Faction members evaluate deeds using a fairly complex formula. This formula is described in detail in the **Deed Evaluation Formula** section near the end of the manual.

If the faction member's **% Judge Parents** value is non-zero, it will not only adjust its affinity to the deed actor but also to the deed actor's parents. For example, if Cat Girl does something nice, it reflects well not only on her but on her Kingdom (her parent faction). Make sure to avoid looping family trees or this will cause an infinite loop.

AUTOMATIC CHARACTER INTERACTION

Love/Hate provides components that allow characters can perform automatic actions based on their Love/Hate values.

Greeting Trigger

Use a **greeting trigger** to make a character play a greeting animation when it sees another character.

To add a greeting trigger, select the character, and then select menu item **Component > Love/Hate > Greeting Trigger**, or **Greeting Trigger 2D** for 2D scenes. The character must have a trigger collider.

In the Animation Triggers list, specify reactions for ranges of affinity values and temperaments.

> Example: In the example above, the animator trigger parameter “Sad” will trigger when the character's affinity to the other character is less than -25. The temperament dropdown is set to *Everything* on this row, which means this line applies regardless of the character's current temperament.

You can also add a script that implements `IGreetEventHandler` if you want the character to do something more when it greets another character. The Scripting section later in the manual explains this in more detail.

Gossip Trigger

Use a **gossip trigger** to allow a character to share rumors when they see another friendly character.

To add a gossip trigger, select the character and then select menu item **Component > Love/Hate > Gossip Trigger** (or **Gossip Trigger 2D** for 2D scenes). The character must have a trigger collider. When another character enters the trigger, this character will check its affinity to the other character. If it's positive, they'll share rumors.

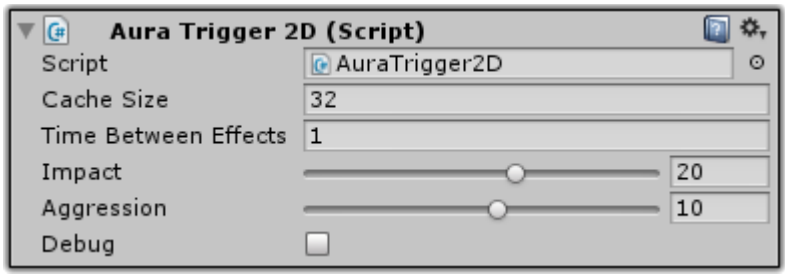
Set **Time Between Gossip** to the number of seconds the character should wait before sharing gossip again with the same friend.

To make a character play an animation when gossiping, add a Gossip Animation component: **Component > Love/Hate > Gossip Animation**.

You can also add a custom script that implements `IGossipEventHandler` if you want the character to do something more when it gossips with another character. For example, you could play a murmuring audio clip or show a cartoon gossip bubble above their heads. The Scripting section later in the manual explains this in more detail.

Aura Trigger

Use an **aura trigger** to affect characters when they enter the trigger area.



To add an aura trigger, select a GameObject with a trigger collider and then select menu item **Component > Love/Hate > Aura Trigger** (or **Aura Trigger 2D** for 2D scenes). This will also add a Traits component if one doesn't already exist.

Set **Time Between Effects** to the number of seconds the aura should wait before applying its effect again to the same character.

Set the values on the **Traits** component.

Impact is the strength of the aura, where a high value applies a stronger effect.

Aggression affects how submissive or dominant the aura makes the character feel based on how their traits align with the aura's traits.

> **Example**: Say you've decided to add an aura to a vampire's lair. The Purity trait is -80, indicating it's an evil place. The Impact is 50, a powerful aura. Aggression is 30, a dominant place. When a devout villager (with a negative trait alignment) enters the lair, he will lose pleasure and dominance, and his arousal will increase. When the vampire enters its lair (with a positive trait alignment), it will gain pleasure and dominance, and its arousal will decrease.

SAVING AND LOADING

Love/Hate is designed to support easy saving and loading with third party save systems. Components don't maintain any special runtime pointers to other components or external data sources, so you can serialize and deserialize any of them without having to link anything back up.

The FactionManager and FactionMember classes provide methods to serialize and deserialize data from strings:

```
public string SerializeToString()
public void DeserializeFromString(string s)
```

To save the state of the FactionManager or a FactionMember, call SerializeToString() and save the resulting string. To load, provide the string to DeserializeFromString().

FactionManager Serialized String Format

Serialized FactionManager data uses this format:

numFactions,
(*for each faction:*) numParents,{parentIDs},numRelationships,
    (*for each relationship*:) {factionID,inheritable,traitValues}

FactionMember Serialized String Format

Serialized FactionMember data uses this format:

factionID,padHappiness,padPleasure,padArousal,padDominance,numMemories,
  (*for each memory*:) memDeedGuid,deedTag,actorFactionID,targetFactionID,impact,
  repetitions,confidence,pleasure,arousal,dominance,shortTermTimeLeft,longTermLeft

Other Save Systems

If you're using the Dialogue System or ORK Framework, see their sections below for instructions on using their save systems.

SCRIPTING

API Reference

The complete API reference is online at: http://pixelcrushers.com/lovehate/api

Namespace

Love/Hate's source code is contained in the namespace `PixelCrushers.LoveHate`.

Add this line to the top of your scripts to use Love/Hate:

```
using PixelCrushers.LoveHate;
```

FactionManager

You can use FactionManager to manage most things, such as affinities and deeds. FactionManager has a lot of methods. Refer to the API reference for details.

You will usually have one FactionManager in your scene, although Love/Hate supports multiple concurrent FactionManagers. In this case, simply assign the desired FactionManager to each faction member. See *Saving and Loading* above for instructions on writing save/load code.

FactionMember

You can override default behavior on faction members by assigning your own methods to these delegates:

CanSee Delegate

If set, the faction member will call this delegate instead of the default visibility method to determine if it can see the actor of a deed.

```
void Start()
{
    GetComponent<FactionMember>().CanSee = MyCanSee;
}

public bool MyCanSee(FactionMember other, Dimension dimension)
{
    // (Return true or false)
}
```

EvaluateRumor Delegate

If set, the faction member will call this delegate instead of the default rumor evaluation method.

```
void Start()
{
    GetComponent<FactionMember>().EvaluateRumor = MyEvaluateRumor;
}

public Rumor MyEvaluateRumor(Rumor rumor, FactionMember source)
{
    // (Return a new rumor based off the source's rumor.)
    // (You can call ApplyRumorImpact() to take care of some default handling.)
}
```

GetPowerLevel and GetSelfPerceivedPowerLevel Delegates

There are two delegates related to power level. **GetPowerLevel** returns the faction member's power level as perceived by others. **GetSelfPerceivedPowerLevel** returns the faction member's self-perceived power level. For example, if a faction member is very insecure, he may have a lower self-perceived power level than his actual power level.

Example for GetSelfPerceivedPowerLevel:

```
void Start()
{
    var factionMember = GetComponent<FactionMember>();
    factionMember.GetSelfPerceivedPowerLevel = MySelfPerceivedLevel;
}

public float MySelfPerceivedPowerLevel()
{
    // Insecure character thinks he's half as good as he really is:
    return GetPowerLevel() / 2;
}
```

PAD

Every FactionMember has a **pad** field. You can modify its values by calling Pad.Modify().

```
void Sleep()
{
    // When sleeping, decrease arousal (3rd parameter) by 10:
    GetComponent<FactionMember>().pad.Modify(0, 0, -10, 0);
}
```

Temperament is a function of the PAD values. You can access it in code by calling PAD.GetTemperament():

```
var temperament = GetComponent<FactionMember>().pad.GetTemperament();
```

It returns an enum (see the table in the Faction Member section above) that corresponds to a specific state such as `Temperament.Exuberant` or `Temperament.Anxious`.

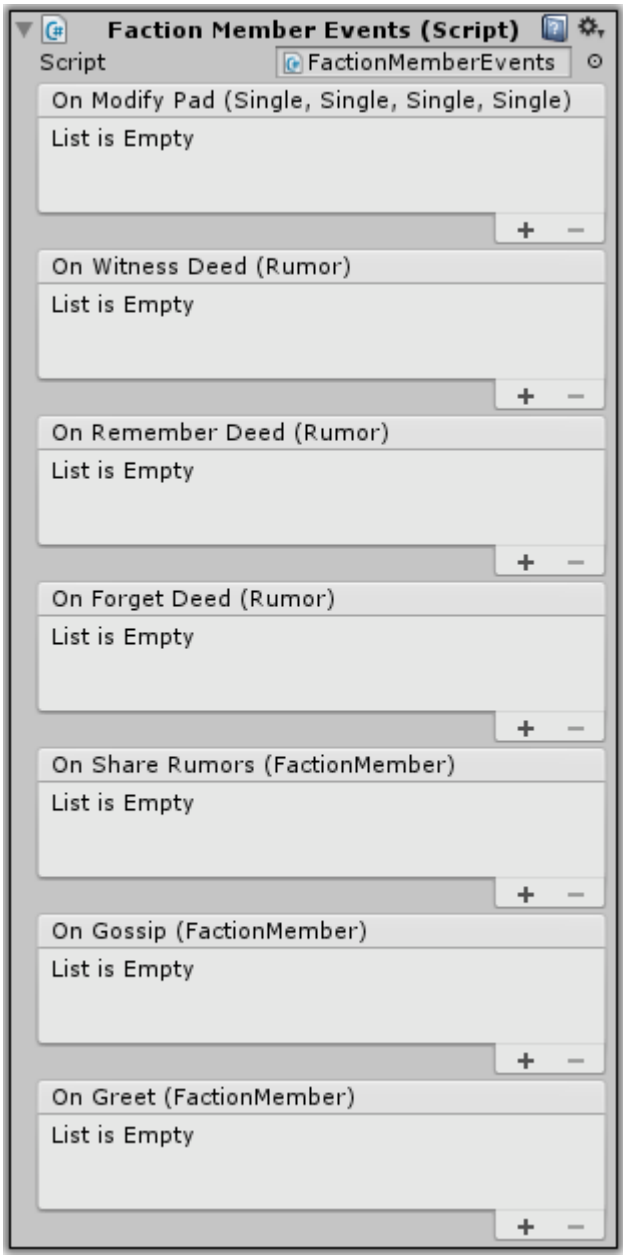
DeedReporter

DeedReporter has one method: ReportDeed(). Call it to report a deed in code. The deed tag should correspond to a deed template in the DeedReporter's deed template library.

```
GetComponent<DeedReporter>().ReportDeed("attack", selectedTarget);
```

Event Handlers

Love/Hate uses the Unity event system introduced in Unity 4.6. If you want to visually assign your own methods to Faction Member events in Unity's Inspector view, add a Faction Member Events component (**Component > Love/Hate > Faction Member Events**).



If you want to automatically tie your methods to Love/Hate events in script without having to assign anything in the Inspector view, add handlers for these events:

Witness Deed Event Handler

This event occurs when a faction member witnesses a deed. In the example below, if the deed displeases the faction member, the faction member will attack the actor.

```
using PixelCrushers.LoveHate;
public class MyDeedHandler : MonoBehaviour, IWitnessDeedEventHandler
{
    public void OnWitnessDeed(Rumor rumor)
    {
        if (rumor.pleasure < 0)
        {
            Say("Hey! That's my friend!");
            Attack(rumor.actorFactionID);
        }
    }
}
```

Share Rumors Event Handler

This event occurs when a faction member shares rumors with another. In the example below, the faction member will stroke its chin as if contemplating the information it just received.

```
using PixelCrushers.LoveHate;

public class MyShareRumorsHandler : MonoBehaviour, IShareRumorsEventHandler
{
    public void OnShareRumors(FactionMember other)
    {
        GetComponent<Animator>().SetTrigger("strokeChin");
    }
}
```

Gossip Event Handler

This event occurs when two characters meet and decide to share rumors. The distinction between this event and ShareRumors is that it's possible for a character to share rumors without meeting another character and gossiping – for example, if the character broadcasts news over a radio. Gossip, on the other hand, is face to face. Your custom handler can play audio, show a cartoon bubble, etc. In the example below, the faction member will play a murmur audio clip to indicate that the characters are talking.

```
using PixelCrushers.LoveHate;

public class MyGossipHandler : MonoBehaviour, IGossipEventHandler
{
    public AudioClip murmur;

    public void OnShareRumors(FactionMember other)
    {
        GetComponent<AudioSource>().PlayOneShot(murmur);
    }
}
```

Greet Event Handler

This event occurs when two characters meet and decide to greet each other. The default greet trigger already plays an animation. You can add a custom handler to do something extra such as play audio, show bark text, etc. In the example below, used for a pickpocket, the faction member appraises what the other character owns. Later in the game, the pickpocket might use this information to choose a victim.

```
using PixelCrushers.DialogueSystem;

public class MyGreetHandler : MonoBehaviour, IGreetEventHandler
{
    public void OnGreet(FactionMember other)
    {
        RememberValuables(other);
    }
}
```

Aura Event Handler

This event occurs when a character enters an aura trigger. It occurs on the aura. You can add a custom handler to make the aura do something extra. In the example below, when a character enters the aura, it instantiates a “Mist” prefab onto the character.

```
using PixelCrushers.DialogueSystem;

public class MyAuraHandler : MonoBehaviour, IAuraEventHandler
{
    public void OnAura(FactionMember other)
    {
        Instantiate(Resources.Load(“Mist”), other.transform.position,
                                            other.transform.rotation);
    }
}
```

Enter Aura Event Handler

This event occurs when a character enters an aura trigger. It occurs on the character. You can add a custom handler to do something extra. In the example below, when a character enters a “bad” aura (one whose traits are negatively aligned with the character's traits), the character draws a weapon to prepare for trouble.

```
using PixelCrushers.DialogueSystem;

public class MyAuraHandler : MonoBehaviour, IEnterAuraEventHandler
{
    public void OnEnterAura(AbstractAuraTrigger aura)
    {
        var traits = aura.GetComponent<Traits>();
        var alignment = Traits.Alignment(traits, other.faction.traits);
        if (alignment < 0) // “Bad” aura from this character's perspective.
        {
                other.GetComponent<Combat>().DrawWeapon();
        }
    }
}
```

Emotional State

If you've added an Emotional State component to a faction member, you can call `EmotionalState.GetCurrentEmotionName()` to get the name of the current emotion, or `EmotionalState.GetCurrentEmotion()` to get the index into the `emotionDefinitions` array.

Accessing Traits

If you've added a Traits component to a non-faction member GameObject, you can get trait values using the `GetTraits(string traitName)` function:

```
using PixelCrushers.LoveHate;

public class VampireAI : MonoBehaviour
{
    // This function checks if an object is something the vampire should avoid:
    public bool IsObjectHarmful(GameObject obj)
    {
        var traits = obj.GetComponent<Traits>();
        return (traits != null) && (traits.GetTrait("Purity") > 50);
    }
    … (rest of script) …
}
```

You can also the static `Traits.Alignment(float[], float[])` function to check how well a two sets of traits align. It returns a normalized value in the range [-1,+1] that indicates how well their extremes match, not how close their values are. If both sets are all zero, the alignment will be zero. If all traits in both sets are at their extremes (+100 or -100) and the sets match exactly, the function will return +1. If both sets are at their extremes and the sets are the exact opposite, the function will return -1.

```
// This function checks if an item is something a character would
// want based on how the item's traits align with the character's personality:
public bool DoesCharacterWantItem(FactionMember character, GameObject item)
{
    var itemTraits = item.GetComponent<Traits>().traits;
    return Traits.Alignment(character.traits, itemTraits) > 0;
}
```

ADVENTURE CREATOR SUPPORT

Love/Hate has built-in Adventure Creator support. To enable it, import this package:

Assets/Pixel Crushers/LoveHate/Third Party Support/Adventure Creator Support.unitypackage

The package will create this folder:

Assets/Pixel Crushers/LoveHate/Third Party Support/Adventure Creator Support

and it will also unpack several actions in:

Assets/AdventureCreator/Scripts/Actions/ActionLoveHate*.cs

Adventure Creator Example Scene

To play the Adventure Creator example scene you must first select **ManagerPackage** in the Example folder. In the Inspector view, click "Assign managers".

In the example scene, the NPCs have hotspots and interactions. To interact, click on an NPC. To keep the example simple, the interaction simply flatters the NPC.

Adventure Creator Actions

The Adventure Creator Support package will add these actions to Adventure Creator:

Action	Description
Check PAD	Checks PAD values.
Check Parent	Checks if a faction has an ancestor or direct parent.
Check Relationship	Checks the value of a relationship trait.
Create New Faction	Creates a new, empty faction.
Destroy Faction	Removes a faction from the faction database.
Modify PAD	Modifies PAD values.
Report Deed	Reports a deed. The actor must have a **DeedReporter**.
Set Parent	Adds or removes a parent from a faction.
Set Relationship	Sets a relationship trait value.
Share Rumors	Shares rumors from one faction member to another.

Remember Scripts for Saving and Loading

To save the state of the faction database, add a **Remember Faction Manager** script to the Faction Manager. To save the state of a faction member, add a **Remember Faction Member** script. Both scripts are available in the menu **Component > Love/Hate > Third Party > Adventure Creator**.

DIALOGUE SYSTEM SUPPORT

Love/Hate has built-in Dialogue System for Unity support. To enable it, import this package:

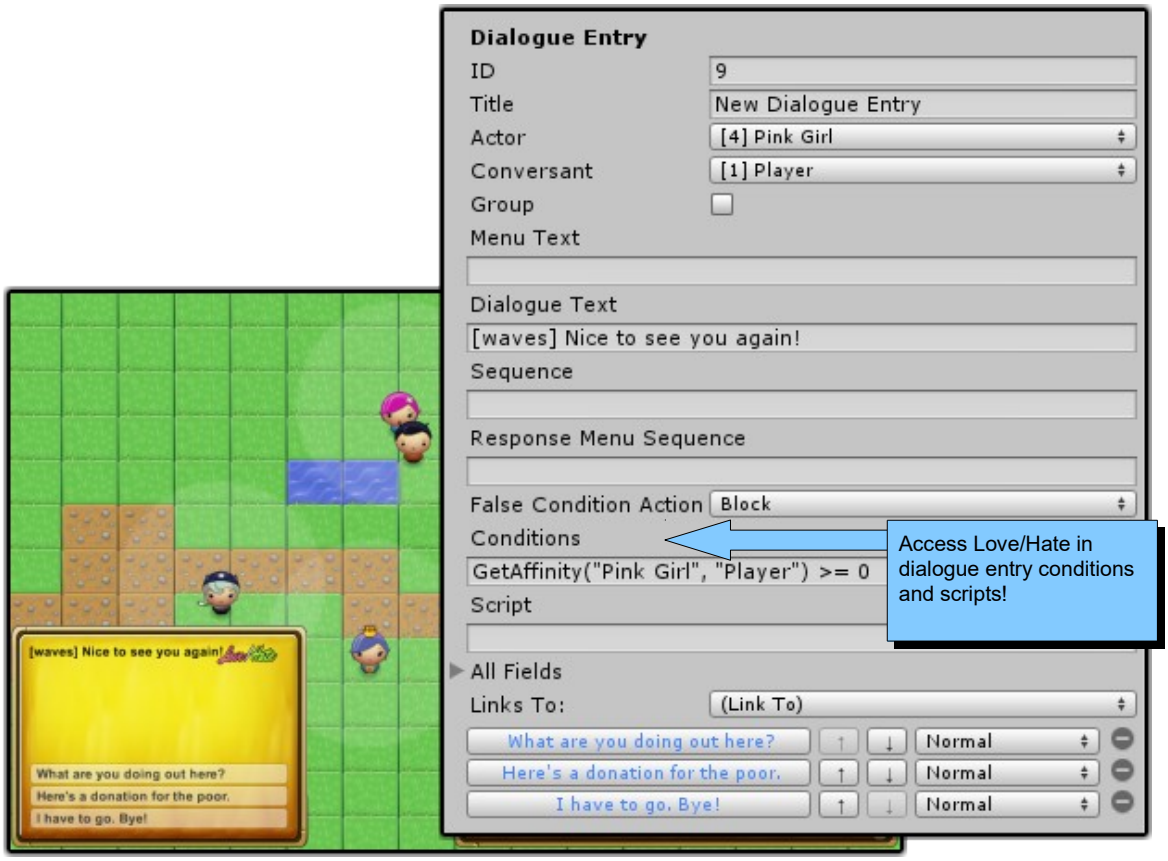
Assets/Pixel Crushers/LoveHate/Third Party Support/Dialogue System Support.unitypackage

The package will create this folder:

Assets/Dialogue System/Third Party Support/LoveHate

Dialogue System Example Scene

The Dialogue System example scene plays just like the original Love/Hate example scene with the addition of a **Talk** button. When you bump into an NPC, click this button to converse.



It also adds **Save** and **Load** buttons on the upper right that demonstrate saving and loading Love/Hate data.

Lua Functions

The Dialogue System interfaces with Love/Hate through a library of Lua functions. To enable the Lua functions, add a **LoveHateLua** component to the Dialogue Manager GameObject (**Component > Dialogue System > Third Party > Love/Hate > LoveHateLua**).

The **LoveHateLua** component adds the Lua functions detailed below, which you can use in conversations' Conditions and Script fields. For examples that use these Lua functions, examine the example scene's dialogue database

GetFactionName(gameObjectName) *[returns string]*

Gets the faction name of a GameObject that has a faction member component.

GetAffinity(judgeName, subjectName) *[returns number]*

SetAffinity(judgeName, subjectName, value)

ModifyAffinity(judgeName, subjectName, dValue)

Gets, sets, or modifies (increments/decrements) the affinity a judge feels toward a subject. For the judge and subject, provide the names of the GameObjects that have faction member components.

GetRelationshipTrait(judgeName, subjectName, traitName) *[returns number]*

SetRelationshipTrait(judgeName, subjectName, traitName, value)

ModifyRelationshipTrait(judgeName, subjectName, traitName, dValue)

Gets, sets, or modifies a relationship trait value a judge feels toward a subject. For the judge and subject, provide the names of the GameObjects that have faction member components.

GetHappiness(nameString) *[returns number]*

GetPleasure(nameString) *[returns number]*

GetArousal(nameString) *[returns number]*

GetDominance(nameString) *[returns number]*

ModifyPAD(nameString, dHappiness, dPleasure, dArousal, dDominance)

Gets or modifies (increments/decrements) the PAD values of a faction member. Provide the name of the GameObject that has a faction member component.

KnowsDeed(nameString, actorNameString, targetNameString, deedTag) *[return Boolean]*

Checks if a GameObject (nameString) with a faction member component knows about a deed committed by an actor to a target.

ReportDeed(actorNameString, targetNameString, deedTag)

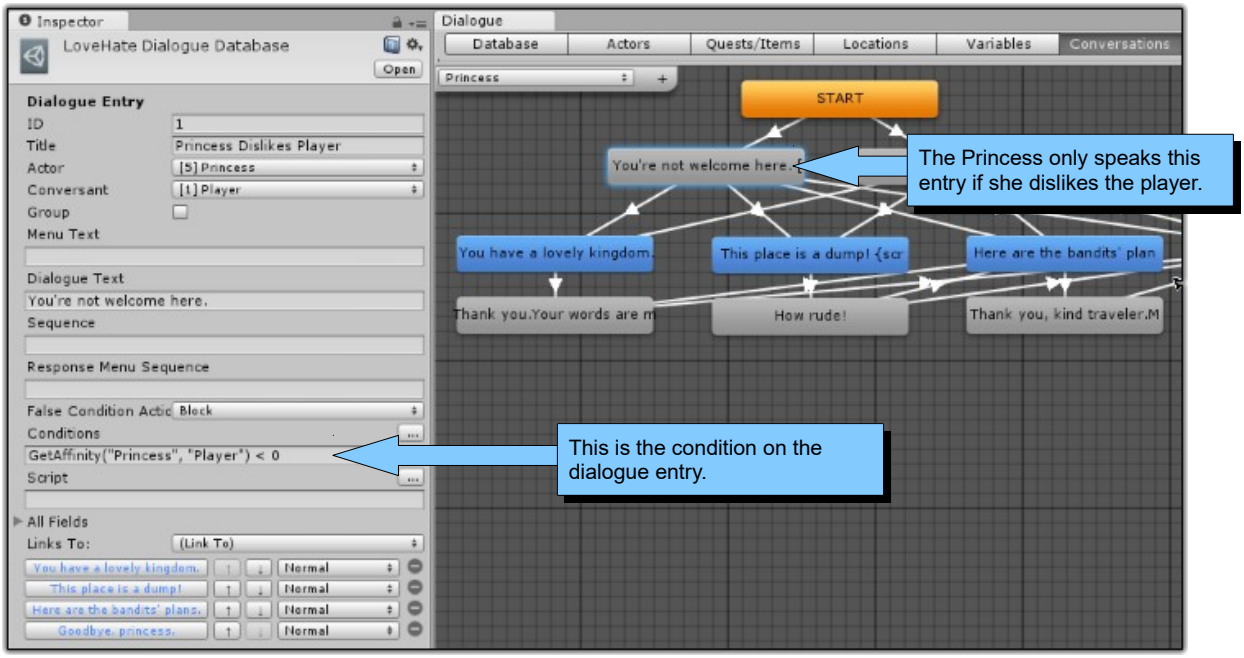
Reports a deed committed by an actor to a target. The actor must have a **Deed Reporter**.

ShareRumors(string actorName, string targetName)

Shares rumors from an actor to a target (one-way).

Example: Branching Conversation Based on Affinity

It's very easy to branch the conversation based on affinity. In the example scene, the Princess says, “You're not welcome here.” if she dislikes the player. This is done by adding a simple **GetAffinity()** condition to the **Conditions** field:



The condition:

```
GetAffinity(“Princess”, “Player”) < 0
```

is only true if the Princess has a negative affinity to the player.

Similarly, you can use the **SetAffinity()**, **ModifyAffinity()**, and **ReportDeed()** functions in a dialogue entry's **Script** field.

Using the Dialogue System to Save and Load

To tie Love/Hate into the Dialogue System's Save System, add a **Persistent Faction Manager** component to the faction manager (**Component > Dialogue System > Third Party > Love/Hate > Save System > Persistent Faction Manager**).

Also add a **Persistent Faction Member** component to each faction member (**Component > Dialogue System > Third Party > Love/Hate > Save System > Persistent Faction Member**).

MAKINOM SUPPORT

Love/Hate has built-in Makinom support. To enable it, import this package:

Assets/Pixel Crushers/LioveHate/Third Party Support/Makinom Support.unitypackage

The package will create this folder:

Assets/Pixel Crushers/LioveHate/Third Party Support/Makinom Support

The package adds these features to Makinom:

- Makinom Faction Manager and Makinom Faction Member components that tie faction managers and faction members into Makinom.
- Schematic nodes for working with Love/Hate in Makinom schematics.

Makinom Example Scene

The Makinom example scene works just like the main example scene except it uses a Makinom dialogue to allow the player to commit deeds. Love/Hate Event Machines on NPCs also log to the console when NPCs witness deeds.

Makinom Setup

To use Love/Hate in a Makinom project:

1. Set up your faction database and deed template library as normal.
2. In your game starter scene, add a Faction Manager GameObject. However, instead of adding a Faction Manager component, add a Makinom Love/Hate Faction Manager component by selecting menu item **Component > Love/Hate > Third Party > Makinom > Makinom Love Hate Faction Manager**.

3. Optional: When preparing your characters (prefabs or scene objects), add a **Love/Hate Event Machine** component by selecting menu item **Component > Love/Hate > Third Party > Makinom > Love Hate Event Machine**. Assign schematics to start when certain Love/Hate events occur (Witness Deed, Enter Aura, etc.).
4. Optional: Add an **Aura Event Machine** to your aura events by selecting menu item **Component > Love/Hate > Third Party > Makinom > Aura Event Machine**. Assign a schematic to play when the aura is entered.
5. Use the **Report Deed** node in your schematics to report deeds to Love/Hate.
6. Use the various **Check** nodes in your schematics to check Love/Hate values.
7. Use other Love/Hate nodes to modify PAD values, change faction membership, etc.

Makinom Love/Hate Faction Manager

The **Makinom Love/Hate Faction Manager** component is a modified version of Faction Manager that integrates with Makinom's save system. It saves the faction database and faction member data.

Love/Hate Event Machine

The optional **Love/Hate Event Machine** component starts schematics when Love/Hate events occur. Add this component to faction members. You can assign schematics to these events:

- Witness Deed: Occurs when the character witnesses a Love/Hate deed.
- Share Rumors: Occurs when the character shares rumors with another.
- Enter Aura: Occurs when the character enters a Love/Hate aura trigger.
- Greet: Occurs when the character greets another using a Greet Trigger.
- Gossip: Occurs when the character gossips with another using a Gossip Trigger.

The *Witness Deed* event sets these global variables before playing the schematic:

- rumorTag (string): The tag of the deed associated with the rumor.
- rumorPleasure (float): The pleasure value for the witness.
- rumorActor (string): The actor's faction name.
- rumorTarget (string): The target's faction name.

Aura Event Machine

The **Aura Event Machine** component starts a schematic when an aura event occurs. Add it to an aura.

Love/Hate Makinom Schematic Nodes

Love/Hate's schematic nodes are in the Love/Hate submenu. They are:

Node	Description
Add Direct Parent	Adds a direct parent to a faction.
Change Affinity	Changes a character's affinity to a faction.
Change PAD	Changes a character's PAD values.
Change Relationship Trait	Changes the value of a relationship trait a character feels for a faction.
Check Affinity	Checks a character's affinity to a faction.
Check Arousal	Checks a character's arousal value.
Check Dominance	Checks a character's dominance value.
Check Happiness	Checks a character's happiness value.
Check Has Ancestor	Checks if a character has an ancestor faction.
Check Has Direct Parent	Checks if a character has a faction as a direct parent.
Check Knows Deed	Checks if a character knows about a deed.
Check Pleasure	Checks a character's pleasure value.
Check Relationship Trait	Checks the value of a relationship trait a character feels for a faction.
Create New Faction	Creates a new, empty faction.
Destroy Faction	Permanently removes a faction from the faction database.
Get Temperament	Gets a character's temperament value (a string).
Inherit Traits	Sets a faction's traits to the values inherited from its parents.
Remove Direct Parent	Removes a direct parent from a faction.
Report Deed	Reports a deed to Love/Hate. The actor must have a **DeedReporter**.
Share Rumors	Shares rumors between two characters.
Switch Faction	Changes a faction member to a different faction.
Use Faction Manager	Sets the faction manager that the faction member should use.

Using Makinom to Save and Load

The **Makinom Love/Hate Faction Manager** component automatically ties into Makinom's save system.

ORK FRAMEWORK SUPPORT

Love/Hate has built-in ORK Framework support. To enable it, import this package:

Assets/Pixel Crushers/LoveHate/Third Party Support/ORK Framework Support.unitypackage

The package will create this folder:

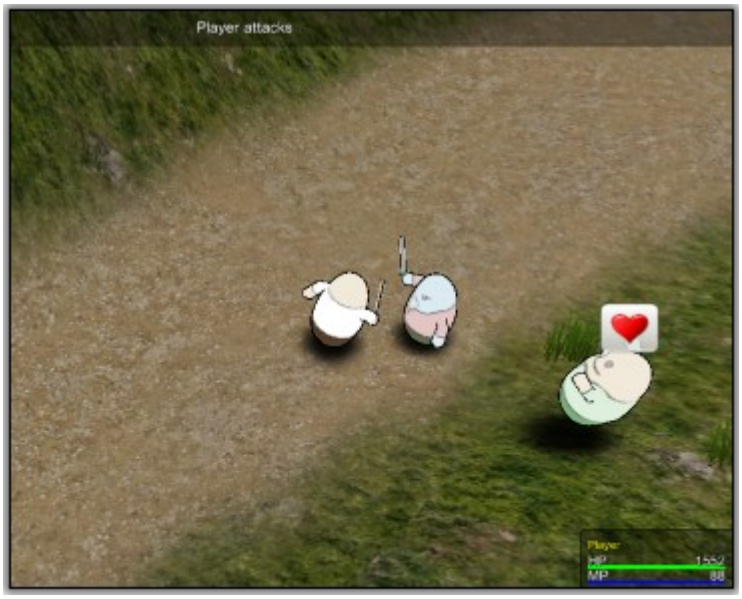
Assets/Pixel Crushers/LoveHate/Third Party Support/ORK Framework Support

The support package for ORK Framework is designed to replace ORK's faction system. You can use both systems in your project, but they won't talk with each other unless you write events to do it.

The package adds these features to ORK:

- **Ork Faction Manager** and **Ork Faction Member** components that tie faction managers and faction members into ORK.
- Event steps for working with Love/Hate in ORK events.

ORK Example Scene



The example scene uses assets from ORK's tutorial game. You must first import ORK's tutorial game before playing the example scene.

If you want to edit the example scene's settings in the ORK Framework editor window, move the ORKProject asset file from Love/Hate's ORK Framework Support folder to Assets/ORK Framework.

The Player belongs to the Love/Hate Player faction, Green Pants belongs to Allies, and Evil Pants belongs to Enemies.

The example scene uses a modified attack event named “attackAndReportDeed” that reports an “attack” deed through Love/Hate. It's assigned to Status > Abilities > 0:Attack > Battle Event 1.

Green Pants (an ally) has an event interaction for the Witness Deed event and also has a conversation that checks affinity to the player.

ORK Setup

To use Love/Hate in an ORK game:

8. Set up your faction database and deed template library as normal.
9. In your game starter scene, add a Faction Manager GameObject. However, instead of adding a Faction Manager component, add an Ork Faction Manager component by selecting menu item **Component > Love/Hate > Third Party > ORK Framework > Ork Faction Manager**.
10. When preparing your characters (prefabs or scene objects), instead of adding a Faction Member component, add an Ork Faction Member component by selecting menu item **Component > Love/Hate > Third Party > ORK Framework > Ork Faction Member**. Assign ORK events to start when certain Love/Hate events occur (Witness Deed, Enter Aura, etc.).
11. Use the Report Deed event step in your ORK events to report deeds to Love/Hate.
12. Use the various Check event steps in your ORK events to check Love/Hate values.
13. Use other Love/Hate event steps to modify PAD values, change faction membership, etc.

Ork Faction Manager

The **Ork Faction Manager** component is a modified version of Faction Manager that integrates with ORK's save system. It saves the faction database and faction member data.

Ork Faction Member

The **Ork Faction Member** component is a modified version of Faction Member. It adds two things:

- Uses the combatant's power level when evaluating power levels for dominance values.
- Starts ORK events when Love/Hate events occur.

Use it connect an ORK event to a Love/Hate event, add one or more ORK **Event Interaction** components to your character. Then assign them to the fields at the bottom of the Ork Faction Member inspector. You can assign these events:

- Witness Deed: Occurs when the character witnesses a Love/Hate deed.
- Share Rumors: Occurs when the character shares rumors with another.
- Enter Aura: Occurs when the character enters a Love/Hate aura trigger.
- Greet: Occurs when the character greets another using a Greet Trigger.
- Gossip: Occurs when the character gossips with another using a Gossip Trigger.

Aura Event Interaction

The **Aura Event Interaction** component starts an ORK event when an aura event occurs.

Love/Hate ORK Event Steps

Love/Hate's ORK event steps are in the Love/Hate submenu. They are:

Step	Description
Add Direct Parent	Adds a direct parent to a faction.
Change Affinity	Changes a character's affinity to a faction.
Change PAD	Changes a character's PAD values.
Change Relationship Trait	Changes the value of a relationship trait a character feels for a faction.
Check Affinity	Checks a character's affinity to a faction.
Check Arousal	Checks a character's arousal value.
Check Dominance	Checks a character's dominance value.
Check Happiness	Checks a character's happiness value.
Check Has Ancestor	Checks if a character has an ancestor faction.
Check Has Direct Parent	Checks if a character has a faction as a direct parent.
Check Knows Deed	Checks if a character knows about a deed.
Check Pleasure	Checks a character's pleasure value.
Check Relationship Trait	Checks the value of a relationship trait a character feels for a faction.
Create New Faction	Creates a new, empty faction.
Destroy Faction	Permanently removes a faction from the faction database.
Get Temperament	Gets a character's temperament value (a string).
Inherit Traits	Sets a faction's traits to the values inherited from its parents.
Remove Direct Parent	Removes a direct parent from a faction.
Report Deed	Reports a deed to Love/Hate. The actor must have a **DeedReporter**.
Share Rumors	Shares rumors between two characters.
Switch Faction	Changes a faction member to a different faction.
Use Faction Manager	Sets the faction manager that the faction member should use.

Using ORK to Save and Load

The **Ork Faction Manager** component automatically ties into ORK's save system.

PLAYMAKER SUPPORT

Love/Hate has built-in PlayMaker support. To enable it, import this package:

Assets/Pixel Crushers/LoveHate/Third Party Support/PlayMaker Support.unitypackage

The package adds several PlayMaker actions categorized under “Love/Hate”.

PlayMaker Example Scene

IMPORTANT: To play the example scene, you **must** first import **PlayMakerUnity2D.unitypackage**, which available at: https://hutonggames.fogbugz.com/?W1150

The PlayMaker Example scene plays just like the original Love/Hate example scene, except the Player has a PlayMaker FSM. When you bump into an NPC, the FSM shows buttons in the top left of the screen for Flatter and Insult. The FSM demonstrates how to get factions, get affinities, and commit deeds using PlayMaker.

PlayMaker Actions

The Love/Hate PlayMaker support package adds these actions to PlayMaker's Actions browser:

Action	Description
Faction Membership	
Add Direct Parent	Adds a direct parent to a faction.
Create New Faction	Creates a new, empty faction.
Destroy Faction	Permanently removes a faction from the faction database.
Get Faction Name	Gets the faction name of a GameObject that has a FactionMember component.
Has Ancestor	Checks if a faction has another faction as an ancestor.
Has Direct Parent	Checks if a faction has another faction as a direct parent.
Inherit Traits From Parents	Sets a faction's personality traits based on its parents, summed or averaged based on the setting in the faction database.
Remove Direct Parent	Removes a direct parent from a faction.
Switch Faction	Changes the faction that a faction member belongs to.
Use Faction Manager	Sets the faction manager that the faction member should use.
Relationships	
Get Affinity	Gets the affinity a faction feels toward another faction.
Get Relationship Trait	Gets the value of a relationship trait that a faction feels toward another faction.
Modify Affinity	Increments or decrements the affinity a faction feels toward another faction
Modify Relationship Trait	Increments or decrements the value of a relationship trait a faction feels toward another faction
Set Affinity	Sets the affinity a faction feels toward another faction.

Set Relationship Trait	Sets the value of a relationship trait that a faction feels toward another faction.
Set Relationship Inheritability	Sets whether a relationship is inherited by children or not.
Emotional State	
Get Happiness	Gets the PAD happiness value.
Get Pleasure	Gets the PAD pleasure value.
Get Arousal	Gets the PAD arousal value.
Get Dominance	Gets the PAD dominance value.
Get Temperament	Get a string indicating the PAD temperament.
Modify PAD	Increments or decrements PAD values.
Get Emotional State	Gets the name of the current emotional state.
Deeds	
Knows Deed	Checks if a faction member knows about a deed.
Report Deed	Reports a deed committed by an actor to a target. NOTE: The actor must have a **Deed Reporter**.
Share Rumors	Shares rumors between two faction members.
Saving and Loading	
Serialize To String	Gets a string representation of FactionManager or FactionMember data.
Deserialize From String	Sets FactionManager or FactionMember data from a string.

PlayMaker Events

To allow your PlayMaker FSM to receive events from Love/Hate, add a **Love Hate Events To PlayMaker** component. Select the FSM's GameObject, and then select menu item **Component > Love/Hate > Third Party > PlayMaker > Love/Hate Events**. Your FSM can listen for these events:

Event	Description
OnWitnessDeed	Raised when the faction member witnesses a deed. Data: • IntData: actor faction ID • StringData: actor faction ID, target faction ID, deed tag • FloatData: rumor pleasure
OnShareRumors	Raised when the faction member shares rumors. Data: • GameObjectData: other faction member
OnGreet	Raised when the faction member greets another faction member. Data: • GameObjectData: other faction member
OnGossip	Raised when the faction member gossips with another member. Data: • GameObjectData: other faction member
OnEnterAura	Raised when the faction member enters an aura. Data: • GameObjectData: aura
OnAura	Raised on the aura when a faction member enters it. Data: • GameObjectData: faction member that entered aura

TRADESYS SUPPORT

Love/Hate has built-in support for TradeSys. To enable it, import this package:

Assets/Pixel Crushers/LoveHate/Third Party Support/TradeSys Support.unitypackage

The package will create this folder:

Assets/Pixel Crushers/LoveHate/Third Party Support/TradeSys Support

Before using the TradeSys integration or playing the example scene, make sure you've added TradeSys's required tags as described in the TradeSys manual.

TradeSys Example Scene

The example scene contains four Love/Hate-enabled trade posts, two Love/Hate-enabled traders, and a player. Each post's prices for a trader (including the player) are affected by the post's affinity to the trader. As you interact with the NPCs, their changing affinity to you will influence the prices in their faction's trading posts.

As you play the scene, watch the trade posts in the in the Inspector view. The prices will fluctuate depending on which trader is interacting with the post. The traders and player are controlled by modified versions of the example scripts provided with TradeSys. The example scripts don't prevent traders from interacting with the same post at the same time. If one trader buys all of a good while the other trader is trying to buy the same good, TradeSys will throw a division by zero error. To avoid this in your own project, make sure only one trader at a time interacts with a post.

TradeSys Setup

Make sure you've added TradeSys's required tags.

In TradeSys, trade posts are vendors or shops that buy and sell goods. Traders move goods between posts by buying from one post and selling to another.

TradeSys has its own faction and group system. Love/Hate doesn't use TradeSys's factions; you can use them separately from Love/Hate to limit the posts that traders visit.

How to set up Love/Hate with a trade post

1. On the **Trade Post** component, tick **Settings > Custom** pricing.
2. Add a **Faction Member** component.
3. Add a **Love/Hate Trade Post** component (**Component > Love/Hate > Third Party > TradeSys > LoveHate Trade Post**). The Love/Hate Trade Post component has a price adjustment curve. This scales the price based on the post's affinity to the trader. For example, if the post's affinity to the trader is 50, and the curve's value at 50 is 0.25, then prices will be discounted by 25%.

LoveHateTradePost exposes the following additional functionality:

- **RestoreOriginalPrices()**: Call to set the post's current prices to their original values.
- **RecordOriginalPrices()**: Call to record the post's current prices as the new originals.
- Dictionary **originalPrice[goodsName]**: This property lets you access the recorded original prices directly.
- **AdjustPricesForFaction(Faction faction, bool buyingFromFaction = false)**: Call to adjust the post's current prices for selling to a specific faction. If buyingFromFaction is true, it instead adjusts the prices for buying from a specific faction. The table below specifies how the post's affinity for the faction affects prices:

Faction	Sell to Faction	Buy from Faction
Friend	Low	Low
Enemy	High	Low

Hot to set up Love/Hate with a trader

1. Add a **Faction Member** component.
2. Add a **Love/Hate Trader** component (**Component > Love/Hate > Third Party > TradeSys > LoveHate Trader**).
3. Before the trader interacts with a post, call the post's `LoveHateTradePost.AdjustPricesForFaction()` method to adjust the post's prices based on its affinity to the trader. The example script TSTraderAI2D.cs demonstrates how to call this method.

How to set up Love/Hate with the player

1. Add a **Faction Member** component.
2. Before the player interacts with a post, call the post's `LoveHateTradePost.AdjustPricesForFaction()` method to adjust the post's prices based on its affinity to the player. The example script TSPlayer2D.cs demonstrates how to call this method.

BEHAVIOR DESIGNER SUPPORT

Support for Behavior Designer is maintained by Opsive. You can download the support package from: http://opsive.com/assets/BehaviorDesigner/samples.php

NODE CANVAS SUPPORT

Support for Node Canvas is maintained by the developer of Node Canvas. You can download the support package from: http://nodecanvas.com/resources/

FACTION DATABASE TEMPLATES

Love/Hate ships with the following faction database templates that you can use as a starting point for your own faction database.

Simple Template

The **Simple** template defines a single trait, **Virtue**. You may find that this single trait, and everything Love/Hate can do with it, is entirely sufficient to give your characters the behavior you want.

RPG Template

The RPG template uses a small set of traits that adequately cover most situations found in typical character-based role-playing games:

Trait	Description	-100 Means	+100 Means
Charity	Kindness, empathy, mercy, and altruism	Greedy, self-centered	Altruistic, self-sacrificing
Integrity	Trustworthiness, virtuous, and scrupulous	Dishonest, sneaky	Morally upright, unwavering
Orthodoxy	Order, law, justice, and conformity	Anarchist, free-willed	Conformist, lawful
Violence	Strength and physical force	Pacifist	Warlike
Wit	Intelligence and cleverness	Simple, unintelligent, wary of intellectualism	Smart, shrewd, insightful

RPG Examples

- **Dim-witted but honorable warrior**: Integrity +80, Violence +50, Wit -80
- **Greedy, sneaky, cowardly wizard**: Charity: -90, Integrity: -80, Violence -50, Wit +80

OCEAN Template

The OCEAN template uses the OCEAN “Big Five” personality model, where +100 indicates strong alignment with the trait and -100 indicates strong disagreement with the trait.

Trait	Description
Openness	General appreciation for art, emotion adventure, and variety
Conscientiousness	Self-discipline, duty, and meeting outside expectations
Extraversion	Engagement with the external world, interacting with people
Agreeableness	Considerate, kind, generous, trusting and trustworthy, helpful
Neuroticism	Tendency to feel anger, anxiety, or depression

OCEAN Examples

- **Cool-headed spy**: Openness +70, Conscientiousness +50, Extraversion +80, Agreeableness +60, Neuroticism -90
- **Xenophobic rural bumpkin**: Openness -60, Conscientousness +20, Extraversion -70, Agreeableness -20, Neuroticism +40

MegaOCEAN Template

The MegaOCEAN template uses a detailed subdivision of OCEAN developed by Stéphane Bura for the Game Developers Conference 2012 presentation, “Emotional AI for Expanding Worlds.” The examples below are also taken from the presentation.

Openness	**Conscientiousness**	**Extraversion**	**Agreeableness**	**Neuroticism**
Imagination	Self-efficacy	Friendliness	Trust	Anxiety
Artistic Interests	Orderliness	Gregariousness	Morality	Anger
Emotionality	Dutifulness	Assertiveness	Altruism	Depression
Adventurousness	Achievement-striving	Activity Level	Cooperation	Self-consciousness
Intellect	Self-discipline	Excitement-seeking	Modesty	Immoderation
Liberalism	Cautiousness	Cheerfulness	Sympathy	Vulnerability

MegaOCEAN Examples

Faction: Shy Person

Openness	Conscientiousness	**Extraversion -2**	Agreeableness	**Neuroticism +1**
Imagination	Self-efficacy	Friendliness	Trust	**Anxiety +1**
Artistic Interests	Orderliness	**Gregariousness -2**	Morality	Anger
Emotionality -2	Dutifulness	**Assertiveness -2**	Altruism	Depression
Adventurousness -1	Achievement-striving	**Activity Level -1**	Cooperation	**Self-consciousness +2**
Intellect	**Self-discipline +2**	**Excitement-seeking -1**	**Modesty +2**	Immoderation
Liberalism	**Cautiousness +1**	Cheerfulness	Sympathy	**Vulnerability +1**

Deed: Seduction

Openness +2	Conscientiousness	**Extraversion +1**	**Agreeableness +2**	**Neuroticism -1**
Imagination	Self-efficacy	**Friendliness +1**	Trust	Anxiety
Artistic Interests	Orderliness	**Gregariousness +2**	Morality	Anger
Emotionality +2	Dutifulness	**Assertiveness +1**	Altruism	Depression
Adventurousness +1	Achievement-striving	Activity Level	Cooperation	Self-consciousness
Intellect	**Self-discipline +1**	**Excitement-seeking +1**	**Modesty -2**	**Immoderation +1**
Liberalism	Cautiousness	Cheerfulness	**Sympathy +1**	Vulnerability

EMOTION MODEL TEMPLATES

Love/Hate ships with **OCC Emotion Model**, an emotion model template that maps the 22 OCC emotions into PAD space. The bottom of the model includes the 8 PAD temperaments as catch-alls in case the character's current PAD values don't fit into any of the 22 OCC emotion ranges.

DEED EVALUATION FUNCTION

This section describes how the default deed and rumor evaluation function works. You can override the default function by assigning the delegate FactionMember.EvaluateRumor.

A *deed* is an act committed by an actor to a target.

A *rumor* is a memory of a deed. Rumor data includes:

- **Source**: The faction member delivering the rumor
- **Action tag**: The type of act (e.g., “attack”, “heal”, etc.)
- **Actor**: The faction member who committed the act
- **Target**: The faction member to whom the act was done
- **Impact**: How harmful (-100) or beneficial (+100) the act was to the target
- **Aggression**: How aggressive this type of act is
- **Traits**: Personality trait values associated with this act
- **Confidence**: How sure the source is that the rumor is true

When a faction member learns of a deed or rumor, it affects how the faction member feels – for example, does the action make it like or dislike the actor? Does the action make it feel happy, sad, excited, submissive?

Faction members can acquire rumors by hearing them from another member or by directly witnessing a deed. When a faction member directly witnesses a deed, it evaluates the deed as if it were sharing a rumor with itself with 100% confidence.

When a faction member evaluates a rumor, it updates the following information:

- Affinity to the actor
- Pleasure, Arousal, Dominance (PAD) values
- Short-term and long-term memory

The Formula

This is how the faction member updates information:

1. Compute my confidence in the rumor, based on the source’s confidence in the rumor and how much I like (trust) the source:

   confidence = *affinity-to-source* X *source’s-confidence-in-rumor*

2. Compute how it affects my relationship to the actor, based on how the deed impacts the target:

   Δ*affinity-to-actor* = *affinity-to-target* X *impact* X *confidence*

3. Adjust Δ*affinity-to-actor* by how well the deed’s traits align with my own:

   Δ*affinity-to-actor* += **|**Δ*affinity-to-actor***|** X *trait-alignment* / *trait-alignment-importance*

4. Adjust Δ*affinity-to-actor* based on my arousal level:

   Δ*affinity-to-actor* += Δ*affinity-to-actor* X *arousal* X *arousal-importance*

5. Adjust my pleasure based on Δ*affinity-to-actor*:

   Δ*pleasure* = Δ*affinity-to-actor*

6. Adjust my arousal based on Δ*affinity-to-actor*:

   Δ*arousal* = **|**Δ*affinity-to-actor***|** X *arousal-importance*

7. Adjust my dominance based on aggression, Δ*affinity-to-actor*:

   Δ*dominance* = −*aggression* X **|**Δ*affinity-to-actor***|**

8. Further adjust my dominance based on the power difference curve:

   Δ*dominance* += *power-difference-curve* X **|**Δ*dominance***|**

You can see these numbers in Unity's Console view by ticking the faction member's **Debug Eval Func** checkbox.

The faction member will only commit the rumor to memory if Δ*affinity-to-actor* is greater than its **Deed Impact Threshold**. If the rumor is a repeat (that is, it has the same actor, target, and tag), the faction member won't add a new memory; instead, it will increment the count on the existing memory and scale down the impact based on the faction member's **Deed Acclimatization Curve**.