Ustoria Script

# Introduction

This is a scripting language designed to be used as part of the Ustoria editor and player. These are high-level tools to allow non-technical people to create stories which can be developed into textual presentations, graphic novels, interactive movies and video games. Ustoria Script will be a way for the author of the stories to specify how their characters respond to events that happen to them at different times. The goal of the scripting language is to be declarative whenever possible and only resort to procedural techniques when declarative techniques do meet the requirements.

# Ustoria Script Language

## Scope

Ustoria Script provides 3 scope levels:

1. Global Scope is where you define things which can be accessed throughout the story.
2. Scene Scope is where you define things which can be accessed within a scene of the story for the lifetime of that scene. Normally, a scene exists from the time when it is created until the time when it is destroyed.
3. Advisor Scope is where you specify one or more actions associated with a concern of a character. You can define objects which are local to this scope and can only be accessed from within this scope.

Objects in any scope can access objects in the same scope or a higher scope.

Objects are usually named and these names must follow the following rules:

* names must start with an upper or lower case letter or "\_".
* the remaining characters in a name can be numbers, letters, or underscores.
* Any variable can have its scope specified by using a fully qualified name where the name of the object is prefixed by the name of the scope and the components of the name are separated by "." characters. For example, to access a variable named "x" in the global scope can be accessed as global.x. You can specify the value in any scope using the notation <scopeName>.objectName. You can also use the keyword "super" to specify the scope immediately above you.
* In general, object names should be unique. Within a single scope, object names must be unique. A lower-level scope can re-define an object with the same name in a higher scope. When this is done, the author must be notified of this to ensure that this was done on purpose and not accidentally. The object with the same name in a higher scope can be disambiguated by qualifying the name with the name of the higher scope.
* An object cannot be the same as one of the reserved words in the scripting language.

## Variables

Variables store the value of something. Variables are named and can be constant so that their values cannot be modified after they are initially assigned.

The following variable types are supported:

* number: an integer or floating point number
* string: a character string that has the capacity for 4 GB of data.
* boolean: true or false
* ValueSet: A set of values as defined by the author.

Variables can be declared to be of a specific type or the type can be declared contextually. To declare a variable contextually, you simply assign it a value and the type of the value assigned determines the type of the variable. Once the type of variable has been assigned, it cannot be changed for the lifetime of the variable. Variables are created in one of the three scopes and their lifetime and visibility is determined by the scope in which they reside.

### Value Sets

A value set is a user-defined variable type. It consists of a set of variable values, each of which has a name associated with it. The values can be distinct so that their value is different from every other value in the set or they can be fuzzy so that some of their values overlap some of the values in an adjacent set. To better understand this, let's look at some examples.

All value sets draw their potential values from a subset of the numbers. Consider the case where we want to represent the animation currently being played by a character. The character could be idle, walking, jumping, or running. We could represent this as:

idle

walking

running

jumping

0

1

2

3

In this case, each of the values has been associated with a number and each of the values is distinct. This works well in the situation where we can only be playing one animation at a time. To create a value set like this you need to specify:

* the names of the values that can be assigned (idle, walking, running, jumping)
* the fact that these values should be distinct

In another case, you might want to specify the decorations displayed on a character in your game. You might show a halo around the character when there is a power-up in action, an optional health bar over the character's head, and a circle on the ground around the character if it has been selected by the player. It is possible that any combination of these decorations could be displayed on a character so the variable has to store any combination of the different decorations, including none at all. This represents a set of values that are stored in the variable and any combination of the individual values can be stored. To create a value set like this you need to specify:

* the names of the values which can be stored (halo, health\_bar, selected)
* the fact that this is a set of values

Another type of value set represents a series of values which is not exact. Consider the health of your character. It could be very healthy, healthy, slightly sick, sick or very sick. None of these values is exact and, in fact, the values overlap one another. The transition from sick to very sick does not happen instantly, but gradually. We can represent this graphically, as shown below.

very\_sick

sick

slightly\_sick

healthy

very\_healthy

-10

0

+10

Each of these values actually encompasses a range of the number line, not just a single value. The transition from very sick to sick is gradual, not abrupt. There is an overlap between the two, when you are more one than the other. This lets you use inexact terms and have the story act in an appropriate manner. As the example above shows, you can map the values in the value set to real numbers. This will allow you to store the health as both numbers and a value set at the same time and you can get or set the values as either numbers or the values in the set.

You can also change the shape of the value curves to show that they take on a fixed value for a period of time and to control how much of an overlap there is between values.

very\_sick

sick

slightly\_sick

healthy

very\_healthy

-10

0

+10

This curve changes how the values are interpreted. At either end of the number line, it shows that you become either very sick or very healthy and stay that way. In the middle, there is a range of values around zero where the value is slightly sick and no other values overlap it. You should define curves that best model the situation in your game.

### Collections

In addition to single variables, you can have multiple variables stored in a collection. For example, if your character had a knapsack containing multiple things stored as strings we could create a collection of strings.

Let’s say we have a collection of strings called “knapsack”. We could add a new string to it by typing the following action:

Knapsack.add “knife”

We could remove an object from the knapsack using:

Knapsack.remove “knife”

We could find out if there is a knife in the knapsack by using:

Knapsack.conntains “knife”

We can remove all items in the knapsack using:

Knapsack.clear

You can determine the number of values in the collection using the notation:

Knapsack.size

When you add an item to a collection, it automatically has an integer associated with it. The first object added is assigned 0, the second 1, etc. You can retrieve the second object added to a collection by using:

Knapsack[1]

Another way of adding an object to a collection is to specify the value it should be associated with when you add it to the collection. For example, you could do the following:

Knapsack.add “weapon” “knife”

Knapsack.add “food” “candy bar”

Then, you could retrieve the food in the knapsack using:

Knapsack[“food”] and it would return “candy bar”.

If you wanted to change the food, you could type:

Knapsack[food] = “meat”

## Rules

Rules are one of the ways in which we can specify how a character can behave in a variety of situations.

Rules consist of two components:

1. A set of variables that can be used to store information that can be used by the rule to store information. This information can be used to determine how the rule responds in particular situations or to store information about the state of the object associated with the rule.
2. A set of concerns that a character might have to worry about. For example, a character might be concerned about whether it collides with another object or not. Each concern will have an advisor that tells the character how to handle that concern.

In a given situation, each character is concerned with various factors in the scene in which the character finds itself. For example, a character might want to do something if it collides with another character while it would do something totally different if the character was shot. When one of these concerns happens, the character will have an advisor which determines how the character should respond to the concern. This information can be captured in a table, as shown below.

|  |  |
| --- | --- |
| Concern \ State | Default |
| Collision | Stop animation  playSound “ouch” |
| Approaches | MoveAround |
| Click | ShowDecoration “circle” |
| Creation | Health = 10 |

Each row in the table shows one of the concerns of the character. The right column shows what action the player is advised to take to handle the concern. The first concern is a collision. This happens when the character collides with another object in the scene. When this happens, the advisor tells the character it should stop the animation it is playing and play a sound file named “ouch”.

The table above only has one column of advice for the character. This can change if the character can be in different states. Suppose the character can be healthy or sick. When the character is healthy and gets shot, he might fight back. On the other hand, if a character is sick and shot, he might run away. To specify these additional states, we need to specify a variable holding the current health of the character and specify the value we want the column to correspond to.

In more complex rules, the rules will depend on the state of the character. For example, if your character is healthy he might attack when approached by an enemy. However, when the same character is sick, he might flee. You could set up rules like the following to handle this situation.

|  |  |  |
| --- | --- | --- |
| Concern \ State | Health == HEALTHY | Health == SICK |
| Approaches | Attack | flee |

Each state is indicated by the value of one or more variables. You could have more complex states like:

Health == HEALTHY && Weapon == LOADED

After you do something to handle a concern, you could change the value of one of the state variables and then a different advisor will be used the next time one of the concerns arises. For example, if your character was shot while healthy, this would make the character sick and change the advisors used to respond to the next concern for the character.

### Concerns

Concerns have various attributes that specify the details of what happened when the concern was raised. For example, if you approach another object, you want to know which object you are approaching, the distance of the approach and the direction of the approach.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Concern | Attributes | | | | |
| Click | Time | Click position |  |  |  |
| Approach | Time | Object approached | Distance to object | Direction to object |  |
| Collide | Time | Object hit | Hit position | Direction to object | accuracy |
| Condition | Time |  |  |  |  |
| Creation | Time | position | orientation |  |  |
| Destruction | Time | Position | orientation |  |  |
| moveStart | Time | Position | direction |  |  |
| moveEnd | Time | Position | direction |  |  |
| animationStart | Time | Animation name |  |  |  |
| animationStop | Time | Animation name |  |  |  |

When you specify a concern, you can optionally specify specific values or ranges for the various attributes of the concern. For example, if you specify just an approach concern, you will be notified when another object approaches you within a preset distance. However, you might want to know if the enemy approaches you within 10 units. In that case, you would specify the distance as less than or equal to 10 and the object approached must be the enemy. Since this has specified conditions for the approach, the concern will be triggered only when all of the conditions have been satisfied. You can specify multiple objects which you can approach and the concern will be raised whenever you approach any of those objects.

The various concerns are detailed below.

**Approach** – this is triggered when the character is approached by another character or object or when the character approaches an object or another character. The direction is a vector that shows the position relative to the character. You can also get this translated to in-front-of, behind, beside, over and under.

**Collide** – this is triggered when the character actually collides with another object. It differs from an approach is that physical contact must happen. Accuracy describes the degree of accuracy of collision detection required. In most cases, APPROXIMATE is good enough. In a few games, ACCURATE can be used but this will slow the game considerably.

**Condition** – this tests the value of one or more variables to determine if they satisfy a condition. For example, you might want to do one thing if health <= 0 while you want to do something different if the health is > 0. Multiple variables can be tested for various conditions and the result combined using AND or OR.

**Creation** – triggered when the object is first created in the scene.

**Destruction** – triggered just before the object is destroyed and removed from the scerne.

**moveStart** – triggered when the character starts to move.

**moveEnd** – triggered when the character stops moving.

**animationStart** – triggered when a specific animation starts.

**animationStop** – triggered when an animation ends.

**rotationStart** – triggered when an object starts rotating.

**rotationEnd** – triggered when an object stops rotating.

### Binding

Rules are nice and let you specify how a character handles various concerns. The question is, to which character do the rules apply? This is referred to as the binding and it binds the rules to the character to which they should apply. We can specify the rules first and then bind them to a character that should obey those rules.

Generic rules can be created in the global scope and the same set of rules can be applied to several characters in different scenes by binding these rules to the characters. This allows several characters to follow the same rules without having to write them again.

There is a problem if you specify that you have a rule for when a character collides with a certain object. What if the object you want to check for collision with exists in one scene but not another? Does this mean that you have to rewrite the rule just because the object you want to check collision with has changed?

To allow your character to collide with a specific object, but not have to specify which object when you write the rules, you can specify that the object should be specified via a variable stored with the rules. Variables stored in the rules form their own scope and can only be accessed by the rules inside the same scope. If you want the variables to be accessible outside the scope, you can specify that the variable is externally readable or externally writable or both. If you do not specify this, no rules other than your own can access the variables inside your rules.

When you bind rules to a character or object, you will be able to specify the value of any variables in the rules. This will let you specify the exact objects in the scene that rules will be applied to.

### Rule Instances

Every time you bind a rule to a character, you create a new version of that rule that has a different set of variables than the same rule applied to any other character. This allows each of the characters to have different variable values so that they can behave differently from each other, even though they are following the same set of rules.

### Specialized Rules

One rule can be a specialized form of another rule. This means that the specialized rule:

* Inherits all of the variables that were defined in the parent rule,
* Inherits all the same concerns and advisors as defined in the parent rule.

However, a specialized rule is not just a copy of the parent rule as it can:

* Add new variables,
* Redefine the advisors for certain concerns,
* Add new concerns and advisors that were not in the parent rule.

By adding something to the parent rule, a specialized rule defines special behavior that the parent rule knew nothing about.

## Message Passing

In some situations one character in a scene will want to send a message to one or more other characters in a scene to alert them to something that happened. The message passing system is provided for this purpose. A message consists of:

* A unique name for the message
* The sender of the message
* The recipients of the message
* The content of the message as a collection of strings

When you create a message you specify its name. You then set the content on the message and then send it to the recipient(s). The recipients could be one other object, a list of objects or all objects which registered an interest in the message.

To receive a message an object must have a rule which says it is concerned about that message. When you send a message to an object, that object must have expressed a concern for than message or it will not receive it. If you ask a message to be sent to all recipients, it will be sent to all objects which have expressed a concern for the message.

## Variable Dependencies

In many cases in stories, the value of one variable depends on the values of one or more other variables. For example, the speed at which a character can run depends on how the health of the character and the speed at which the character is trying to run. The character’s health depends on the amount of damage the character has taken during fights as well as the amount of time since the character last ate. We can capture the health dependencies by noting:

* Health is inversely proportional to the amount of injuries a character has,
* Health is inversely proportional to the time since last meal.

An inverse proportionality means that as the number of injuries increases, the health decreases. In the case of the time since the character last ate, health decreases as the time since the last meal increases. Inverse proportionalities indicate that the value of one variable decreases as the value of another variable increases.

The next thing we need to know is whether both injuries and hunger affect health equally or whether one is more important than the other. In general, injuries have much more effect on your health than does hunger. We can approximate this by saying that injuries account for 80% of your health and hunger accounts for 20%. The total for all variables affecting hunger must add up to 100%.

The last thing we need to know is how the character responds to more injuries and how it responds to more hunger. We might decide that if a character reaches 10 injuries all of the character’s health due to injuries is at zero. We also decide that health decreases directly with the number of injuries. This means that if you have 5 injuries your health due to injuries has been cut in half since 5 is half of the maximum number of injuries.

Hunger is a different story. After we eat, we are not hungry for a while and, as time passes, we start getting hungry. However, we reach a certain level of hunger and don’t really get any more hungry. So the transition to hunger starts slowly, gets faster, and then slows again. This behavior is typical of how we feel about many things. We could say that we measure hunger from 0-10 hours and that it moves slowly from 0 as time passes, moves rapidly in the middle time and more slowly as we approach 10 hours. Feelings like hunger are not directly from our senses but are feelings that are interpreted by the mind.

Sensation is based on a different model. For example, you can hear a pin drop, which is a very low volume sound. On the other hand, you can hear a gun shot, which is thousands of times louder. All of your physical senses behave this way: they have a response to a very small input but the response to an input thousands of times larger is only about 10 times as great. This lets your senses handle a much larger range of inputs than they would otherwise be able to.

When we run, we start out slowly and go faster and faster. The faster we go, the more energy it takes. But going twice as fast does not take twice as much energy, it takes about 4 times as much. This explains why we can only run so fast. Going a little bit faster simply takes more energy that we have, so we reach a maximum speed.

All of this information can be captured in the tables shown below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Variable | Depends On | Proportionality | Dependency | Weighting | Min Val | Max Val |
| Health | Injuries | inverse | Linear | 80 | 0 | 10 |
| Health | timeSinceEat | Inverse | Emotion | 20 | 0 | 10 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Variable | Depends On | Proportionality | Dependency | Weighting | Min Val | Max Val |
| runSpeed | Health | Proportional | Linear | 40 | -10 | +10 |
| runSpeed | runSpeed | Inverse | Exponential | 60 | 0 | 50 |

The different types of dependencies are:

Linear This is when the increase in one variable directly creates an increase in another variable. This models many real-world situations such as the deadliness of a character which would be directly proportional to the number of weapons the character has.

Exponential This corresponds to a situation where a slight increase in one variable results in a much larger increase in the variable which depends on it. As the value of the input gets bigger, the dependent variable gets bigger faster. In the real world, wind acts this way. The force exerted by wind depends on the wind speed by grows much faster than the actual wind speed.

Logarithmic The dependent variable gets a value on a low input value but the dependent variable value grows much more slowly than the value it depends on. Human senses like hearing, vision and touch all work this way.

Sigmoid This describes a dependent variable which grows slowly at the beginning, then increases rapidly and then slows before the maximum value is reached. This is the way many human emotions behave.