A

TAMES

User Manual

Tames Version 1.1

Manual Revision 1.1.1

Contents

1 Introduction

Tames is Unity toolkit that is intended to facilitate architects' creation of dynamic, immersive, and interactive visualisation of their works and research. This is a provisional manual for readers to familiarise them with the concept. Both the toolkit and this manual are still in process of developing and there may be inconsistencies and flaws in either or both.

2 Preparation

For using Tames, you need to have a version of Unity Editor (2021.3.20 and above) installed on your PC. Once you made sure you have the proper version of Unity, you would only need to copy Tames folder in your desired location, ideally on an SSD storage for faster loading and updating. Please note that you need a separate folder for each project, so we recommend keeping the original Tames folder intact to be able to copy it multiple times. Please also mind that despite the low size of the toolkit, Unity will add gigabytes of files to each project.

For the first time opening a project, click Open in Unity Hub and select your folder (the project will appear in the list below it for later use). Unity will take minutes to create the requirements for the project.

Note: In future, Tames will be available as a free Unity asset at its Asset Store. In this case, you would only need to add or remove it from your project.

You can import a number of 3D formats into Unity. Usually models created with BIM tools are converted to FBX format, and then imported into Unity (or first imported and optimized into a 3D modelling application). After importing a file, it is important to check the Read/Write Enabled option so that Tames can read the vertex data of the model. This option is available at the Model tab of the prefab's inspector tab:

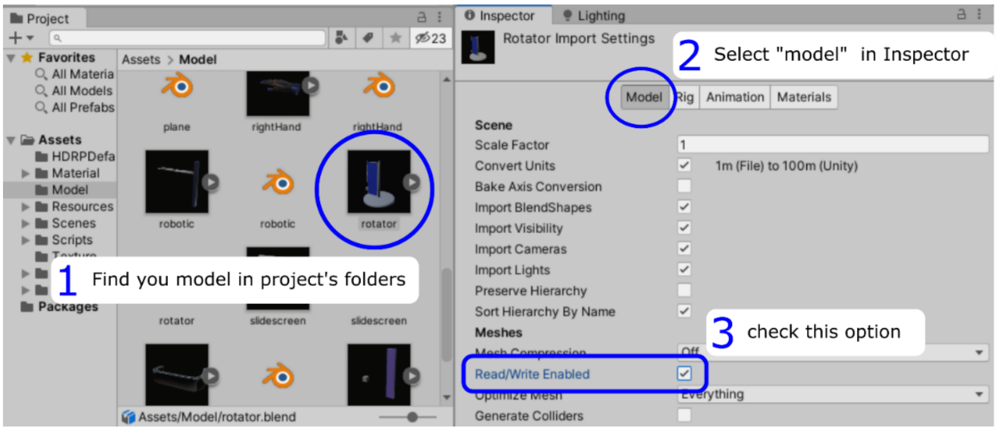


Figure 1.

It is **important** to save you work frequently because Unity does not have an easy autosave feature. Unity is prone to crashing and you may lose component or transform information in case of a crash.

For changes between versions please see 6.1.

3 How does Tames work?

Let's consider that everything has a predefined path of change and movement. For example, a sliding door only slides on a line between two points. Or, a light can only change its intensity from one value to another. Therefore, the only parameter that we need to know for an element's update is its progress on that predefined path, which is simply a percentage or a number between 0 and 1. This value is called the *progress*.

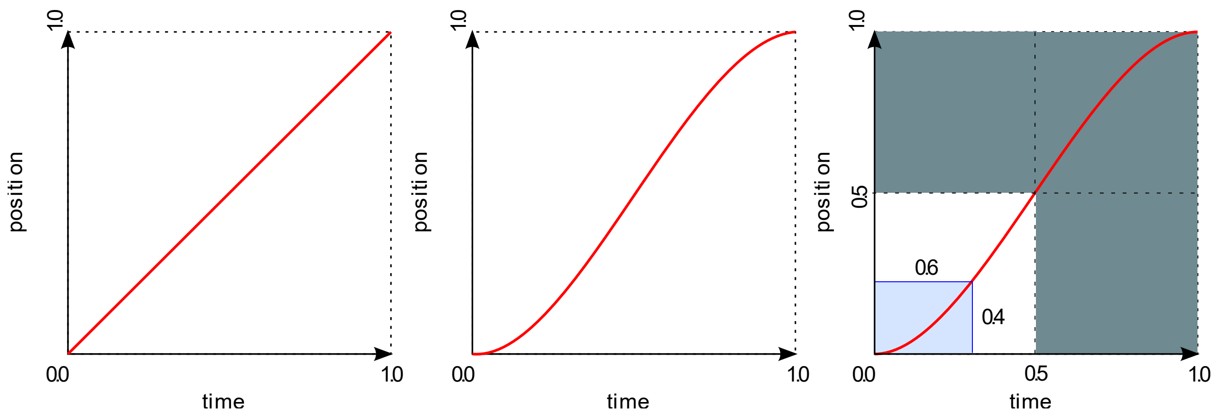
Duration, direction and continuity mode

Of course, Tames needs to know when and how the progress value changes. For example, why should the sliding door open and how fast it should? Each element has several properties that help us clarify these questions for Tames:

- *Duration* of a progress tells us how long (in seconds) it takes for the element to change by one length of its changing path. Its reciprocal value, *speed*, tells us what portion of the progress is passed in every second. This duration is linear and may not feel smooth for 3D objects. For a smooth change, a lerp is used (see below)
- *Direction* of the progress defines if the progress value decreases, stalls or increases as time passes. The direction is applied on the speed value as a mulitiplicative factor of -1, 0 or +1.
- *Continuity mode* defines what happen when the element reaches the end of the path. The sliding door, for example, should stop when it reaches the end of its rail. However, a revolving door may continue to spin after each 360-degree turn. Tames allows three continuity modes: *stop*, *cycle* and *reverse*.

Lerp

Lerping is the change of speed of something, especially when it closes its end of movement. It is very useful to simulate acceleration and deceleration of objects. It works with manipulating the linear association of time and position. For example, imagine an object moves 1m from A to B in 1s. So, its speed is 1m/s. In the left graph below, the speed is constant and so the position changes linearly as time passes. This means, at time 0, the object instantly changes its speed from 0 to 1m/s. In reality, this is impossible as it means an acceleration of (near) infinity. Instead, the object gradually reaches that speed, though in a short time. The graph in the middle shows a more realistic and *lerp*ed correspondence between time and position. In Tames, this is set by two numbers showing how long and how far it takes to reach that linear speed (right graph). These numbers (separated by comma) are between 0 and 1 but are relative to *half* of time span.



The above properties are essential in every automatic change. However, they don't clarify why an element should change to begin with. This is defined by update bases, interaction logics and triggers.

In the examples above, the basis of update was time. When time passes, the element would change based on its speed, direction and continuity mode. However, it is possible to associate update an element based on other elements. For example, imagine an old clock with a pendulum and three hands. The movement of the clock hands depends on the pendulum though they move at different speeds. To establish this relation, we need to tell Tames to update the hands based on the pendulum. So, if we were to right the relationships, it would be like below:

Object	Duration (s)	Continuity	Update basis
Pendulum	1	Reverse	Time
Seconds hand	60	Cycle	Pendulum
Minutes hand	3600	Cycle	Pendulum
Hours hand	43200 (12 hours)	cycle	Pendulum

However, elements are not always continuously changing. Consider that the above clock had a cuckoo alarm that was set on 7AM and 7PM. That cuckoo would come out and in seven times within seven seconds. This means it has a reverse continuity mode, with each turn taking half a second. It should start at hour 7:00:00 and stop at 7:00:07. So, we will have something like this:

Object	Duration (s)	Continuity	Update basis
Cuckoo	0.5	Reverse	Hours

Trigger

But how do we limit the activation to between 7:00:00 and 7:00:07? The answer is by a *trigger*:

0.5833333+0.5834953

The above line is simpler than it looks. Let's begin with the numbers. The first number is 7/12 (hour 7 from 12 hours). The second number represents 7:00:07 out of 12 hours. It tells tames to pay attention to when the progress value of the hours hand, i.e., the update basis of our cuckoo, reaches these numbers. The second important part of the line is the signs, or lack thereof, between and around the numbers. This combination is in fact:

EMPTY*0.5833333+0.5834953****EMPTY***

If you remember earlier, we talked about the three possible directions of changing progress represented by three numbers: -1, 0 and +1. In the trigger line, we defined the directions: 0 or still before 7:00, +1 or positive change after 7:00, and still again after 7:00:07.

An update basis is not the only factor in triggering a change. Remember the sliding door? Its movement is triggered by the presence of people near it. For this we need to define spatial triggers.

Interaction areas or spatial triggers

With interaction areas, the change direction is defined by presence or absence of a person within designated 3D areas, which imitate a sensor in the real world. For example, a cubic area around an automatic door can convert the presence of a person inside it to a positive direction (from closed to open) and the absence to a negative direction (from open to closed). Unlike the previous progress-based triggers, spatial triggers are defined in the 3D models not in text.

An interaction area is a simple 3D geometry (box, sphere, cylinder or plane) that when a person enters it will change the direction of the progress in the attached element. Like the trigger, the direction change is based on a ternary factor (-1, 0, or +1) multiplied on the progress's speed. There are seven ways or *modes* to manage direction with interaction areas. For four modes, a person (**head**) being inside or outside is associated with the direction factor, and for the next three, the moment of a **hand** entering matters:

Mode	Part	Tag	Inside	Outside	On entering
Inside only	Head	In	+1	0	-
Outside only	Head	Out	0	+1	-
Negative inside	Head	Neg	-1	+1	-
Positive inside	Head	Pos	+1	-1	-
On/Off Switch	Hand	1	-	-	Switch between 0 and +1
Two-state switch	Hand	2	-	-	Switch between -1 and +1
Three-state switch	Hand	3	-	-	Switch between -1, 0 and +1

There is also a manual interaction, called **grip** (with tag **g**) that overrides all other interaction and progress properties. It is activated when the hand makes a grabbing gesture inside the area. Then, the element is attached to the hand, with its progress changes as the hand moves while holding the grip.

The last type of spatial interaction is **distance**. Distance interaction works the same as the geometric interaction, with the difference that it can have multiple ranges (other than inside and outside). For example, you can have distances 1, 2, 3, and 4 which means that when the person is closer than 1m or further than 4m, the progress stops. However, it alternates direction between the steps of these distances depending on whether the inside or outside mode is selected. If two distance points are given, the progress proportionally changes to the distance between the two point.

The easiest way to use interaction areas is to create them in your 3D modelling software as simple solid objects and make them children to the object that represents your dynamic element. However, it is important to follow a few naming rules so that Tames can understand the object is an interaction area (alternatively, you can use Marker Components in Unity). The name of an interaction area starts with an underscore (_) followed by its shape (only the first three letter suffice), then another underscore, followed by the tag of its mode. For example, *_box_g* defines a grip area with the shape of a box or *_cyl_in* defines a cylindrical area that activates an element only if you are inside it.

4 Types of elements

Four types of dynamic elements are possible in Tames:

- 3D elements that are object with a moving child object

- Materials with possible changeable properties, including: albedo and emissive colour, U and V offsets of the main and emissive textures, and intensity of emission.
- Lights with possible changeable properties including: colour, intensity and angle (for spot lights).
- Numerical or custom elements that are not visual and only consist of a progress value.

For defining custom elements and converting scene materials, lights into interactive ones, you need to use Unity components. For 3D elements, you can either define them in 3D models or by components.

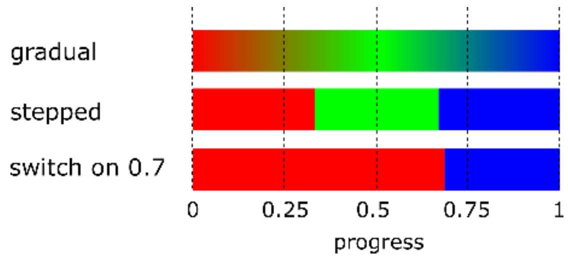
4.1 Changers

Changes in material and lights are based on a structure called Changer. This structure modifies a property by a mode of change and a custom number of steps (minimum of two) which represent the progress value. For example, if a light is turned on, it can be done in two ways:

1. The value of its intensity property should go from step *0* lux to step *100* lux (for example).
2. Or the value of its colour property should go from step *black* to step *white*.

The mode of change defines when and how smoothly the transition between the steps happen. There are three modes:

1. *Stepped* defines an abrupt change. So, the value of the property is always equal to one of the steps.
2. *Gradual* defines a gradual change. The value of the property is calculated based on the progress value's position between the steps.
3. *Switch* creates a threshold before which the property value is the first step and after it is the last.



Three modes of change for a colour property (the steps are red, green and blue).

There is a fourth type, the randomized Flicker that is explained later in the components.

5 Basic 3D modelling for Tames

Tames requires certain characteristics in a 3D model for it to be identified as a dynamic or interactive object. These characteristics can be included in either or both the 3D modelling software and Unity Editor. In summary, to define the movements of a 3D object, we need the movement's path and the base direction of moving on the path. In the 3D model, these requirements can be met by **child** objects whose names starts with *_mov*, *_path*, *_start* and *_end*. Instead of _mov you can use *_head* if you want your element follows a person.

This tell Tames that the object *_mov* moves along with *_path* in a direction defined by *_start* and *_end*. If such marker objects are included, Tames identifies their **parent** object as a dynamic element. The hierarchy of these objects is **important** for Tames to correctly identify them (Figure 5.1).

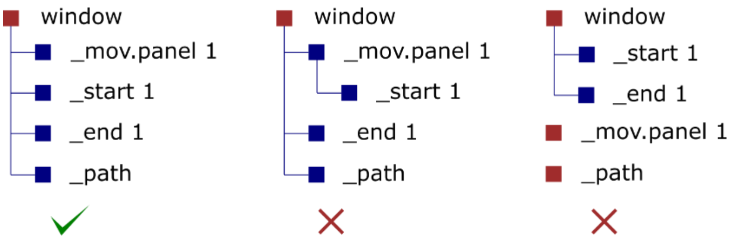


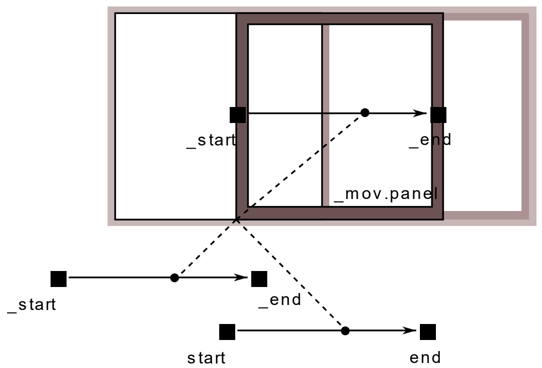
Figure 5.1. All of the marking objects should be child of the same parent (left).

Important Rules

- If you include a path object, the imported model that includes that object should have a readable mesh (see the Preparation).
- Only the initial position of the marker objects, relative to the parent object, is recorded. Later changes to them will not affect the pathing of the latter.
- Each element can only have one marker of each kind (one path, one moving object, etc.). Additional markers are ignored.
- Only the start of the object's name should match with the marker names. So, you can have objects named *_pathsdfsdf* or *_path.123* and they are still valid.
- The path should be a stripe with only one segment on its width. Otherwise, Tames will not recognize it correctly.

5.1 Sliding objects

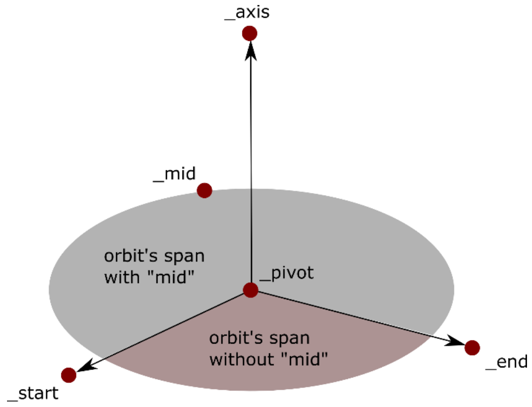
Sliding objects are simplified pathed objects that can only slide on a straight line. Hence, they don't need a *_path* marker but just *_start* and *_end* ones. The moving part slides **parallel to** the vector between start and end **not on** it (Figure 5.2).



Regardless of where the start and end markers are, the moving panel behaves the same as long as their vectors have the same direction and magnitude.

5.2 Orbiting objects

Orbiting objects rotate around an axis. The axis is a vector between *_pivot* and *_axis* markers. A *_start* marker is necessary to indicate the starting angle of the orbiting. An optional *_end* marker determines the span of rotation (its absence means the rotation is 360 degrees). To distinguish between rotation angles under and above 180 degrees, a *_mid* marker is necessary for the latter to identify the larger angle between the start and end points (Figure 5.3).



Using *_mid* marker to define a wide orbiting span.

5.3 Revolving object

Revolving objects rotate around themselves (in contrast to orbiting objects that rotate around an external axis). They are defined in conjunction with pathed or orbiting objects with the marker *_up*. This marker defines the revolving axis which limits their revolution when they are orbiting or moving on the path. In case of pathed objects, this axis is between *_start* and *_up* and for orbiting objects it is between *_pivot* and *_up*. If *_up* is on the same point as the other point of the axis, the object is deemed fixed and it will not rotate. If the *_up* marker is not present, the object will revolve freely (following either the orbit or path's geometry).

5.4 Tracking rotations

If you include *_pivot* but not *_axis*, Tames assumes it is a free rotating object. You can include an *_end* as well which indicates the span of rotation. However, this span cannot be over 90° (it is default if you don't add *_end*). Tracking free rotating objects can only follow people and cannot be controlled by other elements.

The list of all element-defining markers is in the following table:

Marker combinations

Mandatory	**Additional markers**	**Usage**
_mov (or _head), _start	_end	Sliding element
	_pivot, _axis	Orbiting element (full circle)
	_end, _pivot, _axis	Orbiting element (limited angle <180˚)
	_mid, _pivot, _axis	Orbiting element (limited angle defined by _mid)
	_mid, _end, _pivot, _axis	Orbiting element (limited angle defined by _mid and _end)
	_pivot	Free rotating tracker (only with _head), with 90° span
	_pivot, _end	Free rotating tracker (only with _head), with span defined by _end (max 90°)
	_end, _path	Pathed element

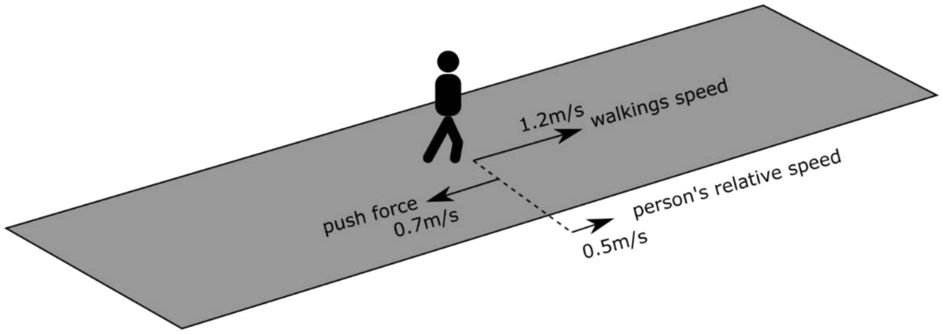
In all non-sliding and non-free rotating combinations, you can include an *_up* marker.

5.5 People's movement

You can limit the movement of people with walkable surfaces. A walkable surface is part of a 3D object that limits the movement of camera on it. When they are imported, their upward-facing surfaces are detected and stored as the navigable area of the space. To define an object as walkable you need to add a Marker Walk component to it (See 5.3).

To walk smoothly in the space, the walkable surfaces must be attached on the plan and be close to each other on vertical sections (less than 30cm level difference) on the section, otherwise, the movement between them will be invalid and impossible. Of course, if the walkable objects move their attachment and proximity may change and alter the walkability on their edges.

You can assign a pushing force to a surface. A pushing force is a velocity vector that affects a person's default moving vector. Pushing forces are useful to simulate walkways and escalators.



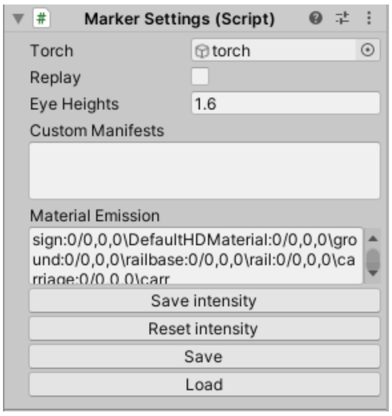
Defining pushing forces works similar to defining moving objects, with only differences in the marker names (we add an "f" in the beginning of the marker name's text). Instead of *_path*, *_start*, *_end*, *_pivot* and *_axis*, we would have *_fpath*, *_fstart*, *_wend*, *_fpivot* and *_faxis*. We can have sliding, rotating and pathed pushing forces, however, the usage of the markers is different. You **cannot** define the markers with components in this version.

For sliding and pathed push, *_fstart* and *_fend* still indicate the direction of the push, but their distance represents the velocity (in m/s) or the force of the push. In rotating push forces, there is no start and end, but only *_fpivot* and *_faxis* that represent both the rotation axis and its angular velocity by their distance (each unit would be 360 degrees per second). The direction of rotation is determined by the direction of the axis (clockwise if you look towards the axis' direction).

6 Unity Components

Unity components are used to define and adjust various dynamic and interactive features. Please note that if you modify the 3D model, the component information may be lost if the hierarchy or name of its attached object is changed.

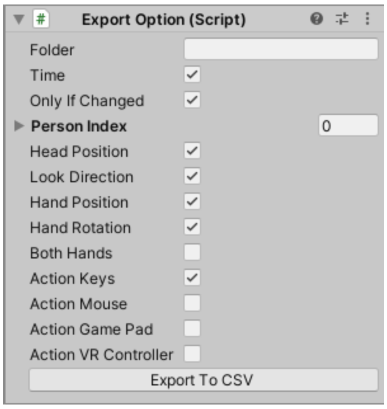
6.1 General settings



A Setting component is added by default to the root object *interactives*.

- **Torch**: the game object assigned here will act like a headlight (so, to work you need to add a light objecct here). You can switch the torch on or off with C key.
- **Replay**: in a **future** version checking this will make Tames to replay and re-enact saved sessions.
- **Eye heights**: separated by commas, you can put various eye heights (i.e. the camera's Y difference to the nearest walking floor). You can switch between them by pressing Z during the Play session.
- **Custom manifests**: advanced feature not covered in this manual.
- **Material Emissions**: do not edit this field see the next item:
- **Save** and **Reset Intensity**: an issue in Unity is that most changes to materials in the Play mode are also applied in the Editor. While for most cases, this is not an issue, for emission intensity or brightness, it causes the material exponentially getting brighter. This is because Unity combines intensity and color, and there is no way for Tames to separate them when the Play mode starts. Therefore, it thinks the existing color is the base color, and adds intensity to it. To prevent this issue, you can freeze the emissive color of materials in their current values in the Editor by the Save Intensity button. Tames will automatically reset the colors at the start of the Play mode to the last time they are frozen. The Reset Intensity button will reset all the material intensities to the saved status. When you add a new material or change an existing one, rememeber to press this button before pressing Save Intensity. Otherwise, you may save the last change during the Play mode.
- **Load** and **Save**: [has not been tested in the last two revisions] When Tames versions change, there is a possibility of changes in components and loss of data. These two buttons allow to save the older components and load them after updating Tames. First, you need to Save your current component (you only select the folder; the file names is chosen automatically based on date and time). Then after replacing the Tames files with its recent update, you click on Load and choose the saved file. If you objects and materials are not changed, Tames will update the components in their original place.

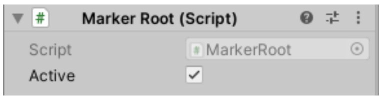
6.2 Exporting movements and interactions



You can export the session to Tames own format (Tames Frame Record or TFR) and comma delimited files (.csv). For Tames own format, you need to press Ctrl+S **while in the play mode**. A file (named based on date and time) would be saved in the folder specified in the Export Option component attached to the "interactives" root object. If no folder is specified, a file dialog opens letting you choose a folder (the session pauses). For converting the saved TFR file, you need to click the Export To CSV button when **not** in the play mode. An file dialog opens in which you choose the TFR file. A CSV file will be created that is named based on the date and time of converting (not of saving the original TFR file).

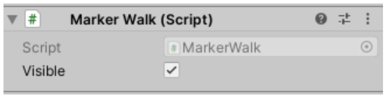
- **Folder**: the default saving folder for the TFR files.
- **Time**: the time stamp of the frame, since the beginning of the session (exclusive of the paused time).
- **Only if Changed**: only exports frames that are different from their previous frame in the items selected below it.
- **Person index**: currently unavailable (this will be available in future for multiplayer sesssions.
- **Both hands**: select this option only when using VR devices.

6.3 Interactive elements



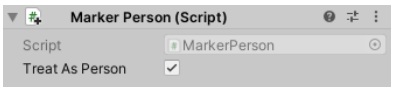
When importing interactive elements, they should be children or descendants of a game object with a Marker Root attached to them, and have its **Active** property checked. Otherwise, they are ignored by Tames.

6.4 Walkable surfaces



Walkable surfaces are defined by adding a Marker Walk component to their game objects. The Visible field determines if they should be shown or not during the presentation.

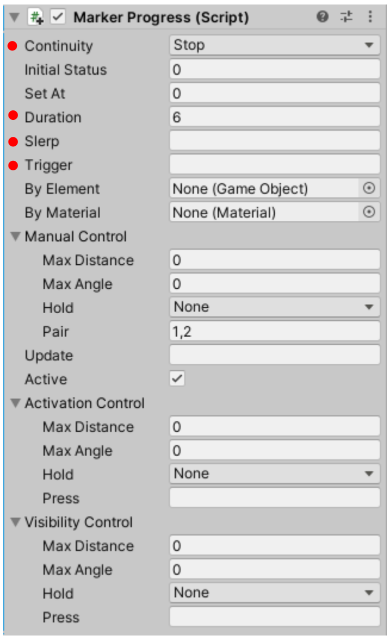
6.5 Non-human "people"



It is possible to treat objects as people (or their head, to be more accurate). For this we attach a Marker Person to that object. To quickly toggle between the treatment we can check or uncheck Treat As Person property.

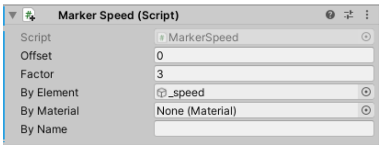
6.6 Progress

The Marker Progress component controls most of the progress of an element. Its fields are (the red dots indicate realtime changes affect the elements):



- **Continuity**: the continuity mode (Stop, Cycle, or Reverse) of the progress.
- **Initial status**: this tell Tames that when the model is loaded in Unity, where the progress initially was at.
- **Set at**: this sets the progress value at the start of Play mode
- **Duration**: the duration of progress in seconds. A negative duration means the progress will have the same duration/speed as its parent. A zero duration means it will exactly follow its parent progress value.
- **Slerp**: the lerping of the progress change.
- **Trigger**: the trigger of the progress. If left empty or erroneous, it will be ignored.
- **By Element and By Material**: sets the update basis of the element based on another element.
- **Manual Control:** sets the manual control (with input devices) for this element (if this is set, every other update is neglaected). To have a valid manual update the **Pair** field should be filled with at least one valid pair of inputs (see Appendix 2). The **Hold** field should is the auxilliary input (e.g., shift) to be hold for the control. If **Max Distance** and **Angle** are set to values above zero, the manual control will be activated when the user is positioned and orientated within those values (in meters and degrees respectively).
- ~~**Update:** the update basis of the element, if it is not defined by element or material. If the update is set to Manual, here the keys should be typed. For acceptable input names see Appendix 2.~~
- ~~**Show By**: an input key that switches the visibility of the element once pressed.~~
- **Active**: if the element is active when the play mode begins
- **Activation and Visibility Control**: They set a manual input for controlling the activation and visibility of the element. They work the same as Manual Control, with only the difference that their **Press** field should be filled by a set of valid single inputs.
- ~~**Activate By**: an input key that switches the active status of the element.~~

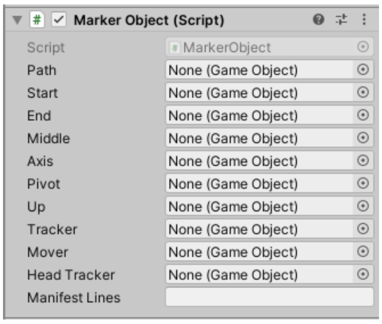
6.7 Variable speed



The Marker Speed component sets a range of speed for an element based on another element.

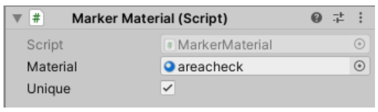
- **Offset**: the default speed in progress per second (i.e. when the base progress is 0). If this is set negative, the default speed will be the element's own speed (set by Duration). Offset of zero means the element will stop at the base progress of 0.
- **Factor**: the factor multiplied to the base progress. The final speed is calculated by the following: *Speed = offset + factor * base progress.*
- **By Element**, **Material** or **Name**: the base element.

6.8 Creating 3D elements



The Marker Object component allows you to set the necessary markers for defining interactive objects in Unity (instead of a 3D modeling application). While the naming requirements do not apply in this mode, the presence of certain markers (e.g., *start*) is the same as the import-based 3D elements. If the component is attached to an object that can pass as an element because of its 3D model, the markers in the component are prioritized in case of duplicate markers.

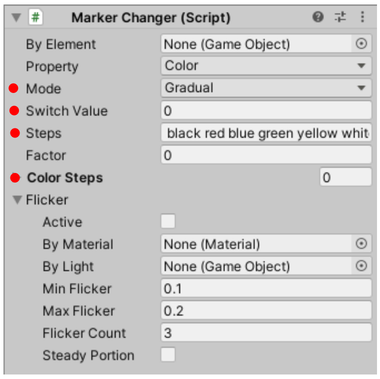
6.9 Materials and lights



The Marker Material component declares that a material is dynamic. It doesn't matter what object this component is attached to as long as it is a child of an active Root element.

- **Material**: the dynamic material (it should have at least one instance on descendants of *interactives* root object).
- **Unique**: if there are more than one instance of that material, copies of them are created to be treated individually. By doing that, the parent or update basis of each instance is automatically set to the closest dynamic ancestor of its holding mesh (if none found, to *time*).
- ~~**update (by element**, and **by name**): for non-unique materials, this sets the update basis of the material.~~

Lights do not require a designated component for them but there should be a Marker Progress component attached to them.

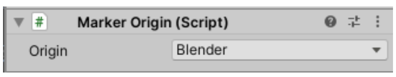


The properties of lights and materials are set by Marker Changer components (multiple components are allowed. Red dots indicate that real-time changes will affect the elements):

- **By Element**: the element that updates this property
- **Property**: the property that should change.
- **Mode**: the changing mode (*stepped*, *gradual* or *switch*).
- **Switch value**: if mode is set as *Switch*, a value between 0 and 1 must be written here to work as the switch threshold.
- **Steps** and **Color Steps**: the list of steps (separated by space). For the accepted color names and custom colors see Appendix 1. Alternatively, you can pick colors in Color Steps (if you use this mode, the text-based steps won't be read)
- **Factor**: use this number to set the base emissive intensity for the emissive color of a material (it doesn't apply to any other property and will be ignored if there is a changer with *Intensity* as its property).

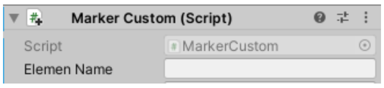
You can created a flickering property by the Flicker sub-property of the changers. Each flickering property needs its own Flicker. It is important to continue flickering the Progress's continuity mode is set to *Cycle* or *Reverse*:

- **Property**: the flickering property. A Changer for this property should exist.
- **By Material** and **By Light**: the flickering of this property will match that of the selected material or light (if they have a the **same** flickering property). The remaining items (below) will have no effect on the flickering if a parent material or light is set.
- **Min** and **Max Flicker**: the minimum and maximum length of each flicker (relative to the *duration* of the element). If you have multiple Marker Flicker's only the first one's Flicker values and counts are used.
- **Flicker Count**: the number of flickers.
- **Steady Portion**: if this is selected there will be no flickering in a continuous portion of progress. The length of this portion is equal to:
  *(1 – min * count) / 2*



Because of an issue in Unity, lights imported from Blender do not pass their children transforms correctly. Therefore, Tames need to correct this at the start of each Play session. If you have such a light (for example with an interaction area attached) in a Blender file, you need to use a Marker Origin component on the Blender's file's main object in the scene to let Tames know the model was created in Blender.

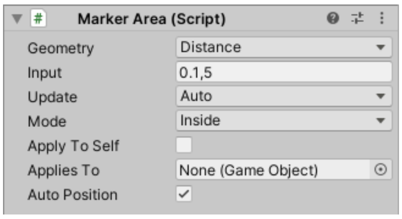
6.10 Custom elements



Custom elements are defined by adding a Marker Custom (the object should not be defined as interactive by other components or model, and should not contain light). For using a custom element as parent of another element, it's best to use its containing game object as the By Element property of the latter's Marker Progress:

- **Element name**: the name of the custom element (can be used as update basis of other elements).

6.11 Interaction areas



Interaction areas are better to be defined in the 3D model of their associated elements, but if this is not possible, you can add a Marker Area component to each element:

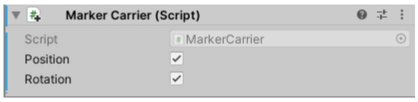
- **Geometry**: the intended shape of the interaction area (it should match the actual shape of the area). The options are Box, Cylinder, Sphere and Plane.
- **Input**: if the area is Switch mode, the key input here will act as its manual switch. For acceptable input names see Appendix 2.
- **Update**: how the area should change by time. *Auto* mode is recommended.
- **Mode**: the mode that the interaction affects its attached element.
- **Apply to self**: Use this option if you want the area to apply to its game object (that would be an element) rather than be a child of the element.
- **Applies To**: attaches the holder of this component as an area for a game object (that should be an interactive element). If no object is selected here, the area is attached to its holder's parent game object.
- **Auto Position**: automatically position the area at the game object containing it.

If Marker Areas are not attached to an interactive object, and their Applies To field is unassigned, Tames creates a custom parameter element at their place. This feature is especially useful for controlling a large number of elements which work based on proximity.

6.12 Camera carriers

A camera carrier is an object which defines the position and/or rotation of the camera. This object is defined by a Marker Camera. During the play mode, you can switch between carriers and the actual camera by pressing C:

- **Position**: associates the position of the camera with this object. Once the carrier is active you can no longer walk. When the carrier is switched back to the actual camera, you will be moved back to its position.
- **Rotation**: associates the rotation of the camera with this object. This option does not work during VR sessions. Once the carrier is active you cannot rotate the camera manually.



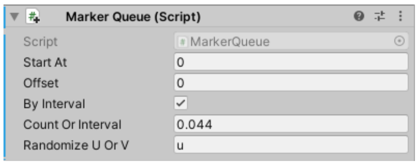
6.13 Cycled object sets

A cycled object set is an element that controls a collection of objects that iterate on a path. A sushi train or a luggage carousel are examples of this type of element:

- **Offset**: the distance between the last and first object on the path, relative to path’s length.
- **Item Names**: the name of 3D objects to be cycled.
- **Children Of**: *all* children of the selected game object will be considered as cycled objects**.**

6.14 “Queued” object sets

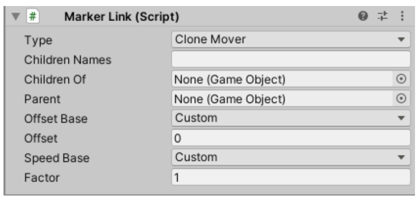
A queued object set is an element whose moving part is cloned with an interval and all the clones move together. This is like the cycled objects with the difference that you only need to define one moving object. An example of this type of element is an escalator:



- **Offset**: the first obejct will be placed at this point (a number relative to the path’s length).
- **By Interval**: if checked, the cloned objects will be placed by an interval (relative to the path’s length)
- **Count or Interval**: the count of clones or the distance between them (if By Interval is checked, the count is calculated automatically).
- **Randomize U or V**: To make the clones less monotone, you can randomize the UV offset of its materials by either u or v (or x or y)**.**

6.15 Linked objects

Linked objects are ordinary objects which turn into dynamic objects based on their link to a predefined interactive object. Currently, Tames only allow 3D object to be linked. There are three ways to create a linked object: a full clone, a mover clone and a motion clone (mover link). A full clone creates a copy the whole of element with all its children (whether moving or not) and interaction areas and progress. The created copy is located and orientated by the linked object. A mover clone is

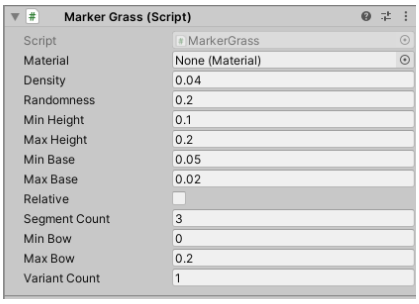


similar to the former but the none-moving children are not cloned. In both types, the clones can act independent of the original object. A mover link only applies the movement of the object to the linked objects. Therefore, the linked objects move as the original changes. The linking is defined by a Marker Link component.

- **Type**: the type of linkage (clone mover, clone everything and link mover)
- **Children names**: the name of linked objects. Only to be filled if the component is attached to the original object.
- **Children of**: linked objects are all the children of this gameobject. Only be filled if the component is attached to the original object
- **Parent**: The game object corresponding to the original object. Only use this on the linked objects.
- **Offset base**: how the offset of progress value for the clones is calculated. The options are *Custom*, *Random*, and by *Parent* (same as original).
- **Offset**: the value of offset (only works if the offset base is set to *Custom*.
- **Speed base**: how the speed of the clone's progress is defined. The options are same as above.
- **Factor**: a value that is used to calculate the duration of clone's progress. This works differently for each *Speed base* type. For *Custom*, the value will be the progress' duration. For *Random*, a random duration is set between *Factor*/3 and *Factor**3. For *Parent*, the duration is set to the parent's duration, multiplied by *Factor*.

6.16 Grass

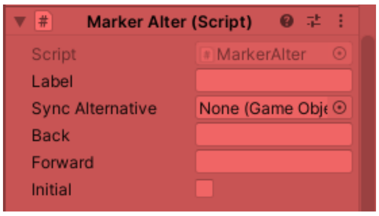
It is possible to create grass-covered surfaces by Tames. The standard procedure is to use Unity's terrain but Tames' grass may be handier for those who are not familiar with Terrains. The position of grass pieces are calculated based on a grid, that is defined by the average distance between adjacent plants. The grid is created on a flat horizontal plane and then projected vertically on the designated surface. The use can define the dimension range of the leaves, their bending, segment count and material. These are set in Marker Grass component.



- **Material**: the grass leaf material. The material can have different variants of grass which are arranged horizontal after each other (so their U values are different). Tames randomly select one section of the material's texture for each leaf.

- **Density**: the distance between each piece of grass (in metres).
- **Randomness**: sets how randomly the grass are planted. The randomness of 0 means the plants are placed exactly on the grid points. Higher numbers indicate the maximum distance (relative to the density value). For example, 0.5 means the possibility 50% distance to the grid points.
- **Min** and **Max Height**: The minimum and maximum possible height of a leaf (in meter). The height for each leaf is calculated randomly.
- **Min** and **Max Base**: The minimum and maximum possible width of the base of a leaf (in meter). The leaves are pointy so their tip's width is always zero.
- **Relative**: Checking this means the base width of a leaf is calculated relative to its height. So, a 15cm-high leaf cannot have the max. base width if the max height is 20cm. Instead, its max. possible base width will be 75% of the max. base value.
- **Segment count**: the number of segments in the Y axis of the triangular outline of a leaf. The value of 1 means a plain triangle. Higher numbers create a realistic bent appearance but also increase the computational load.
- **Min** and **Max Bow**: determines how bent the leaves are. The value is relative to the height of leaves.
- **Variant count**: the number of variants of grass leafs. To make this work properly, the material texture should be accurately divided into this count, horizontally.

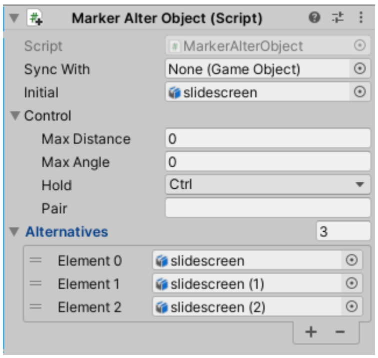
6.17 Altering objects



[Depreciated. See the next image below]
You can alter objects in the scene by Marker Alter components added to each alternative.

- **Label**: the label for the set of alternatives. Objects with the same label in their components are considered part of the same collection.
- **Sync Alternative**: the object is considered as part of the sync object here.
- **Back** and **Forth**: the inputs to change to the previous and next alternatives.
- **Initial**: set this object as the initial alternative.

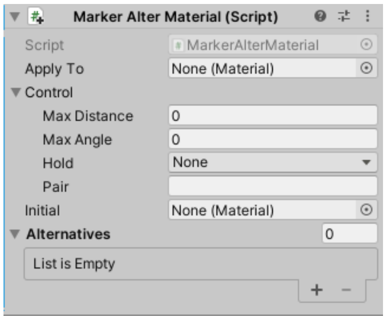
You can alter objects in the scene by Marker Alter Object (MAO) components added to an object.



- **Sync With**: if this is assigned, the object is treated as part of an alternative defined with another MAO. In this case, the rest of the properties will not apply.
- **Initial**: The default alternative (it should be one of the elements in the Alternatives list).
- ~~**Back** and **Forward**:~~
- **Control:** The input control for selecting the previous and next alternative. To have a valid manual update the **Pair** field should be filled with at least one valid pair of inputs (see Appendix 2) with the exception of mouse button. The **Hold** field should is the auxilliary input (e.g., shift) to be hold for the control. If **Max Distance** and **Angle** are set to values above zero, the manual control will be activated when the user is positioned and orientated within those values (in meters and degrees respectively).
- ~~**Activation Distance:** You can only toggle between alternatives if you are within a distance from the active alternative. If set at 0 or smaller, the distance will not matter. This is useful for when you have multiple sets of alternative which you want to control separately, so you won't need to define a separate input keys for each.~~
- ~~**Activation Angle:** You can only toggle between alternatives if your looking direction is within a certain angle (degrees) from the active alternative. If set at 0 or smaller, it will not matter.~~
- **Alternatives:** the list of alternatives.

6.18 Altering materials

You can alter the material of an object by Marker Alter Material components added to an object under the root interactive object. Please note that all of the materials in this component should be assigned to an object in the scene (so that Unity have already loaded them).

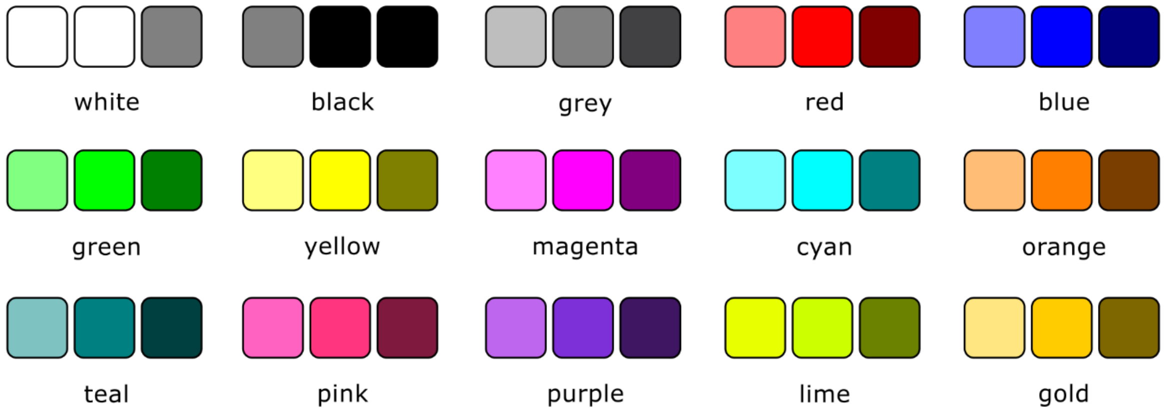


- **Apply to**: the altered material. If the material is dynamic, the dynamic changes still apply to it but combined by the appearance of the alternatives.
- ~~**Back** and **Forth**:~~
- **Control**: see Control in the last marker.
- **Initial**: the initial alternative (should be either none or one of the defined alternatives.
- **Alternatives**: the list of alternatives.

7 Appendices

7.1 Colors

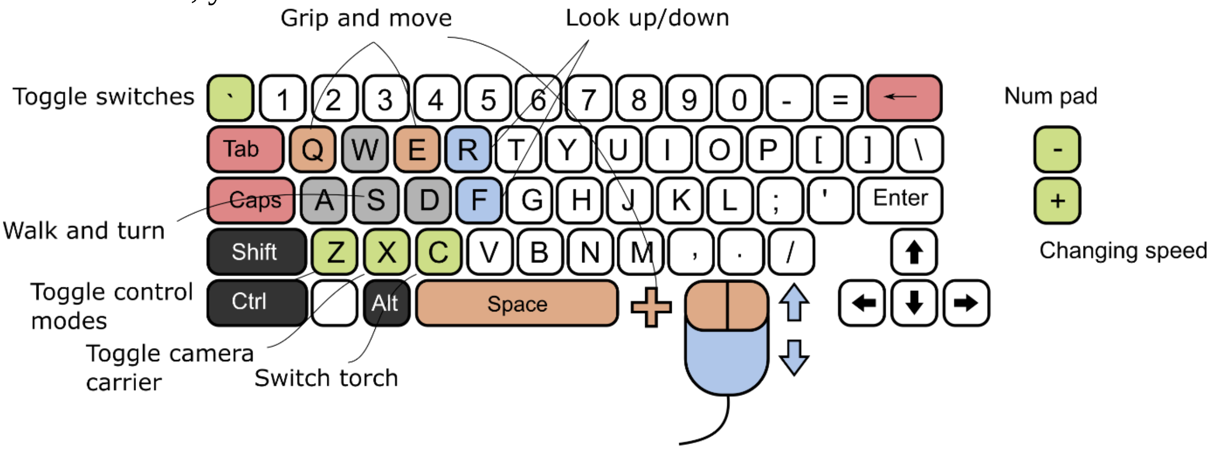
Colors can be defined by their names or their RGBA values. For names the acceptable colors are shown in the figure below.



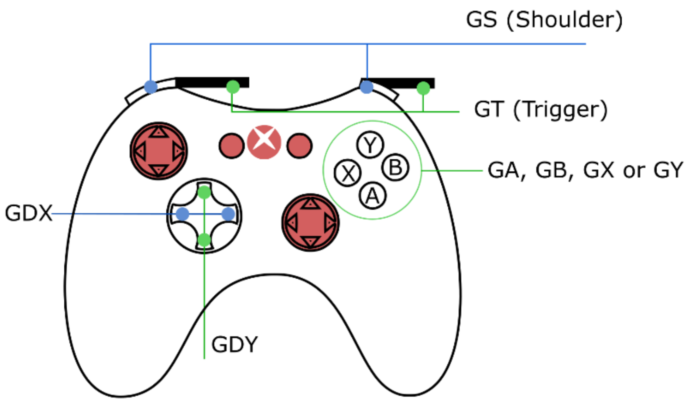
Alternatively, you can define colors with their RGB components, separated by comma. The fourth component will act as the alpha channel for a material's base color (only if the material is transparent). It does stably not affect other properties (in earlier versions, the fourth component denoted the intensity of light and material emission, but they are not debugged in recent updates and may not work correctly).

7.2 Inputs

Some keys are reserved and cannot be used as inputs. The figure below shows the key map of Tames. Only white or black keys can be assigned to new input fields. However, the black keys can only be added to mouse controls (in a format like *button*). For most key, their label is used in Tames. However, for *comma*, *period*, *semi(colon)* and *quote* you need to write their names and as well for the arrows (*left*, *right*, *down*, *up*). For mouse button, you would use *button*.



You **cannot** use mouse buttons for single press. For paired keys (in manual update in Marker Progress), you need to define two keys, separated by comma (e.g., *p,q*). For mouse button you should just use *button* and Tames understands it's a paired control. You can also use a predefined label for paired direction arrows (*hor[izontal]* and *vert[ical]*).



For a game controller, the options are limited. The controller's sticks are reserved for movement and gripping and the Start button is reserved for toggles (it can combine with different trigger buttons as in the Table below). Other buttons can be set on their own or in combination with triggers (trigger buttons cannot be used on their own), using the trigger's and button's labels (figure above).

Trigger	Function
None	Toggle switches
Left	Toggle camera carrier
Right	Switch torch
Both	Toggle control mode

Three buttons (GDX, GDY, and GS) are pairs by default and can be used for manual updates. A pairing shortcut is also possible with X and B buttons (GXB) and Y and A buttons (GYA). The dafault paired buttons can be divided into their individual controls with adding their direction acronym L(eft), R(ight), D(own) or U(p), depending on the buttons (e.g. GDXL means when left trigger is hold and the left button on the DPad is pressed).