A

TAMES

User Manual

Tames Version 0.9

Manual Revision 1

Contents

1 Introduction

Tames is Unity template project that is intended to facilitate architects' creation of dynamic, immersive, and interactive visualisation of their works. The motivation for developing Tames was to minimise a designer's need to use programming when creating such visualisations. The name *Tames* is based on the template's original main class TameObject which itself was based on Unity's main class of GameObject.

This is a provisional manual for readers to familiarise them with the concept. Both the template and this manual are still in process of developing and there are inconsistencies and flaws in either or both.

2 Preparation

For using Tames, you need to have a version of Unity installed on your PC (see 8.1). Once you made sure you have the proper version of Unity, you only need to copy Tames template 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 template folder intact to be able to copy it multiple times. Please also mind that despite the low size of the template, Unity will add around 1GB of files to each project.

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

IMPORTANT

After opening the project, find *SampleScene* in *Assets\Scene* folder and double click on it to open. It is very likely that the project is opened with three missing components, which you would need to add manually:

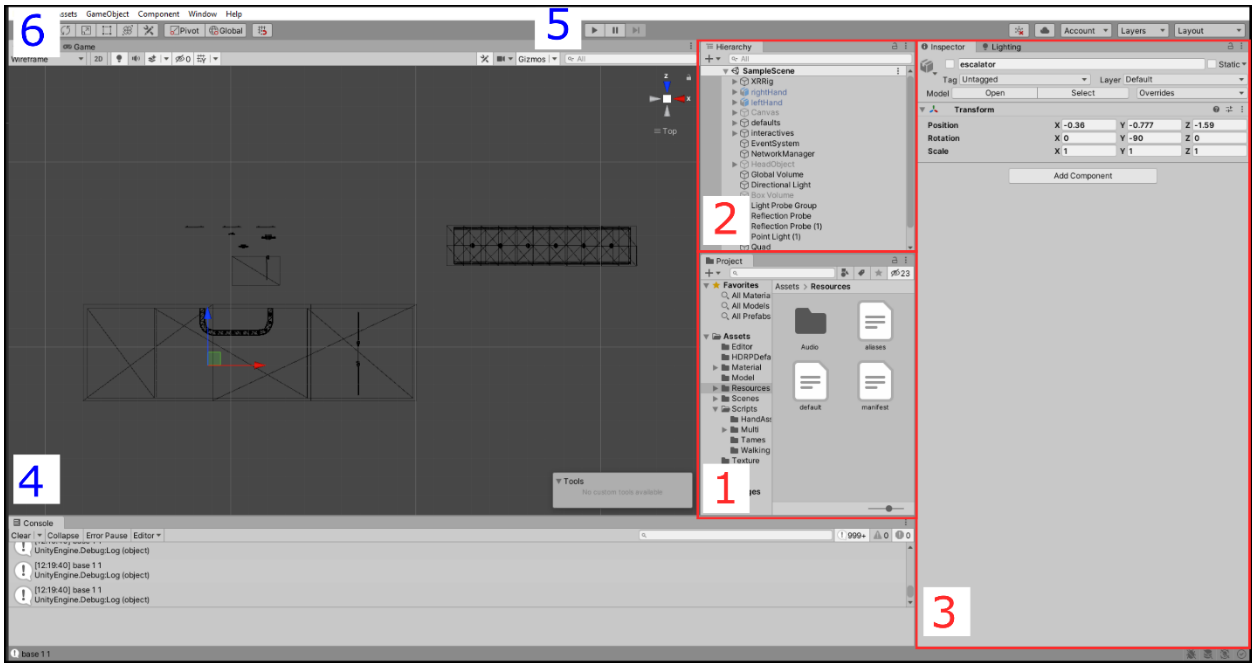
- Click on XRRig in the Hierarchy. In the Inspector panel, you will see a warning for a missing component. Click on the small circle at its right and type "Main". In the list, find "Main Script" and select it.
- Then click on NetworkManager in the Hierarchy. Like the previous one, write the name of the missing component ("Network Manager") and select it from the list.

The current version of Tames is only tried with Valve Index VR headsets. For using this headset you would need to install Steam and its VR add-on, in addition to setting up your VR space (it should be possible to use any SteamVR compatible headsets after this). Please follow Steam's guide.

3 Interface

Tames does not have an interface of its own and depends on Unity Editor. Although the whole of Unity Editor is relevant to any project, Tames is mostly concerned with three panels (Figure 3.1, the layout of your Unity Editor is customizable and may differ from this figure):

1. Project Explorer: the Project Explorer is similar to a file explorer customised for Unity and it shows all relevant files inside the Assets folder of the unity project. Among the folders, the only directly used by Tames is the Resources folder where manifest file(s) are stored. Your models should also be imported your desired folder in Assets folder.
2. Object Hierarchy: the Hierarchy lists all objects used in the active scene. These objects can be Unity's own reserved objects, some setting objects for using the VR devices and lighting, and two objects called defaults and interactives. The last two objects contain all of your interactive elements. Anything outside them is not processed by Tames. Therefore, it is important to include your interactive objects or anything you wish Tames refer to under one of these two objects (preferably interactives, as defaults is for pre-defined objects).
3. Inspector: when you select an object in the Hierarchy, scene or explorer, their properties are shown on the Inspector panel where you can edit them.



escalator
Tag Untagged
Layer Default
Position X -0.36 Y -0.777 Z -1.59
Rotation X 0 Y -90 Z 0
Scale X 1 Y 1 Z 1

[12:19:40] base 1 1
UnityEngine.Debug:Log (object)
[12:19:40] base 1 1
UnityEngine.Debug:Log (object)

Figure 3.1. Unity Editor

Some other important components in the Unity Editor are the Scene (4), the play button (5) and the transform buttons (6).

4 Concept of progress

Tames visualization is based on an associative and linear progress that defines the position or status of a dynamic element on its path or constraints. Therefore, the progress value is in essence the percentage of the path that the element has progressed at. Hence, if we want to manage the changes in an element, we need to manage its progress value.

For this purpose, Tames allows setting of three parameters for changing of a progress: speed, direction, continuity.

1. The **speed** of the progress change is the speed by which the element can progress 100% of its path's length. Reciprocally, it is the duration of progressing this length (in seconds).
2. The **direction** of the progress change is a ternary factor (-1, 0 or +1) multiplied to the speed of change. This will determine if the progress change is backward, still or forward.
3. The **continuity** of a progress pertains to when the progress reaches to one of its ends (0 or 1). For example, a clock's hand will continue in cyclic mode after it reaches 12, while its pendulum will bounce back though in both cases the direction change (time) is forward. Tames considers three continuity modes: **stop** (no change after reaching the end), **reverse** (bouncing back in the opposite direction), and **cycle** (keeping the direction but moving to the other end).

4.1 Connecting progresses

In real life, many progresses are interdepending and connected. In the grandma's clock, the pendulum's bouncing back and forth dictates the rotation of the clock hands, recursively. To simulate a clock, therefore we need to update each hand with its "parent" basis, all up to the pendulum. For example, in the lines below (which can be put in Tames), we define two objects pendulum and seconds (hand) with respective durations, proper continuity modes and updating basis:

```
Object pendulum
        Duration 1
        Reverse
Object seconds
        Duration 60
        Cycle
        Update pendulum (this line means the seconds is updated by pendulum)
```

However, something important is missing here. That would be the effect of pendulum's direction on seconds hand's direction: if the former's direction reverses at each bounce, the seconds' direction should as well, as it is tied to the former's updating. However, this does not happen in Tames because changes are applied not by the normal 0-to-1 progress but by an infinite **raw** progress. As time moves forward, the pendulum's raw progress moves forward too. So, after 75 seconds, its raw progress would be at 75. Similarly, the seconds' raw progress would be at 1.25 (as each 60s is one length of its progress). However, their normal progresses are calculated based on their respective continuity modes (Figure 4.1) and are thus used to update their actual positions.

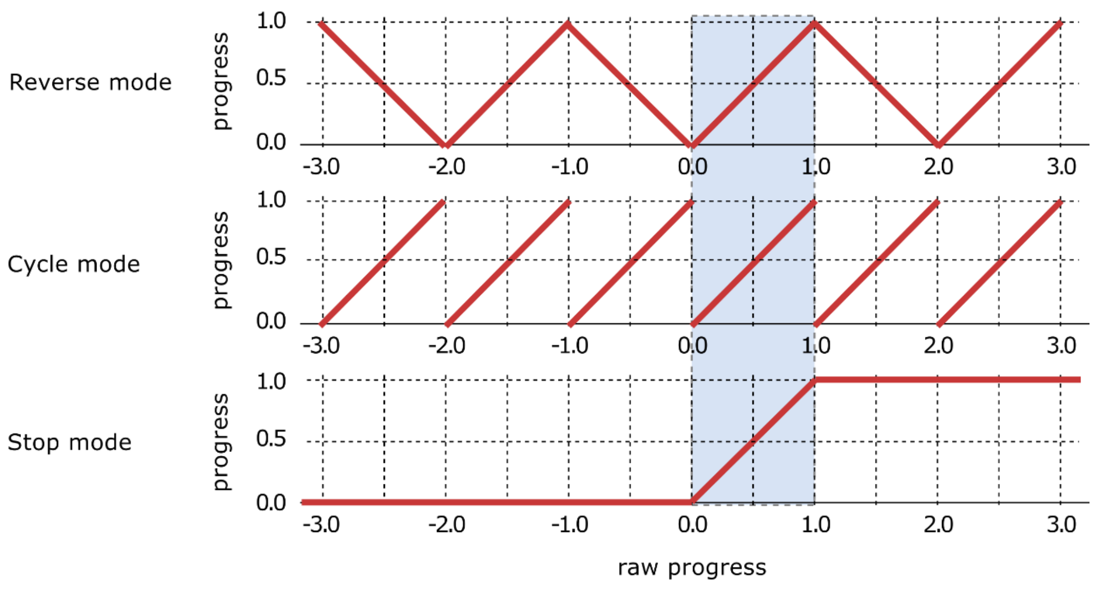


Figure 4.1. Effect of different continuity modes on the progress value.

Considering the concept of direction in the previous section, it would apply to the raw progress, and then the latter's effect on the normal progress is calculated.

4.2 Triggered progresses

Now imagine that a room's temperature is an element with a progress (where its 0-1 range represent temperatures between 0°C and 40°C). We want to visualize a fan heater that automatically turns on when the temperature is under 10°C (or 0.25 progress value). Considering the fan's movement is cyclical, lines below will define the updates like the previous example:

```
Object fan
    Duration 0.1 (10 rotations per second)
    Cycle
    Update temperature
```

The problem with the above lines is that the fan would move if temperature changes, regardless of the temperature's value. However, we want the fan spins indefinitely if the temperature is under 10°C and stops if it is above. In Tames this is done by a trigger. In the line below, Tames understands that when the update basis (temperature) is under 0.25 of its **normal** progress, it should set the direction of the fan's raw progress to positive (+) and if it is over that value, it should stop:

```
    Trigger +0.25
```

4.3 Spatial triggering

Tames also triggers directional changes based on spatial relations between an element and a person (or the camera’s position). 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 lines.

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 naming **child** objects starting with *_mov*, *_path*, *_start* (or *_from*) and *_end* (or *_to*). 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). In Unity Editor, you need to add a MarkerObject component to your **parent** object, in which you can select the moving object, path, and the direction markers, regardless of their names.

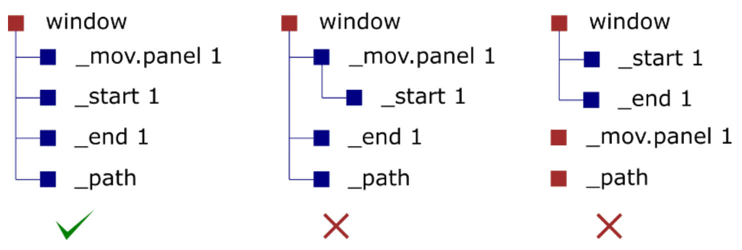


Figure 5.1. All of the marking objects should be child of the same parent (left).

IMPORTANT If you include a path object, the imported model that includes that object should have a readable mesh (see 8.2.18.2.1).

IMPORTANT Only the initial position of the marker objects, relative to the parent object, is recorded for the sake of identifying a dynamic elements. Later changes to them will not affect the pathing of the latter.

IMPORTANT Each element can only have one marker of each kind (one path, one moving object, etc.). Additional markers are ignored.

IMPORTANT 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.

IMPORTANT The path should be a stripe with only one segment on its width (Figure). 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. It is *IMPORTANT* to know that the moving part slides **parallel to** the vector between start and end **not on** it (Figure 5.2).

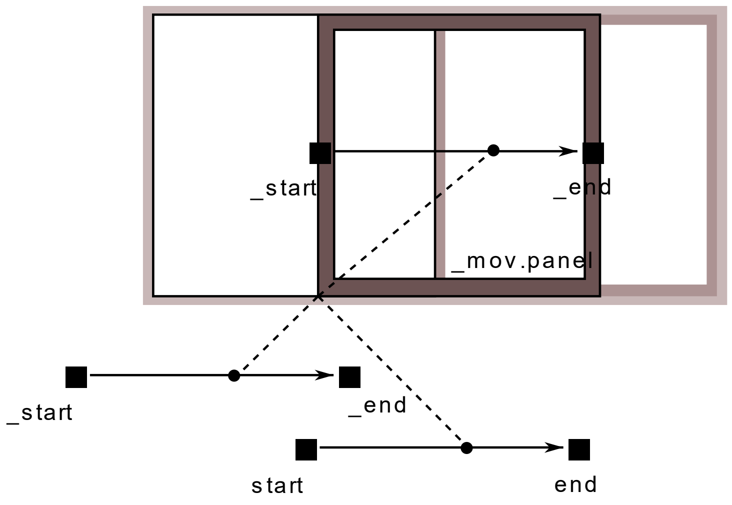


Figure 5.2. Regardless of where the start and end markers are, the moving panel behaves the same as long as the their vectors have the same direction and magnitude.

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).

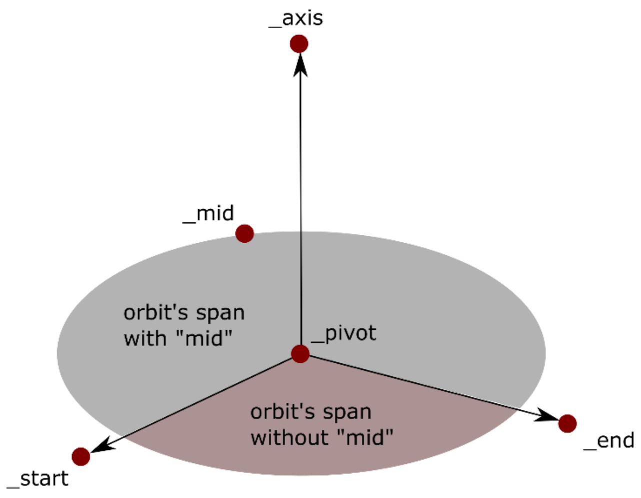


Figure 5.3. Using *_mid* marker to define a wide orbiting span.

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).

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

Table 5.1. Marker combinations

Mandatory	**Additional markers**	**Usage**
	_end	Sliding element
	_pivot, _axis	Orbiting element (full circle)
	_end, _pivot, _axis	Orbiting element (limited angle <180˚)
_mov, _start	_mid, _pivot, _axis	Orbiting element (limited angle defined by _mid)
	_mid, _end, _pivot, _axis	Orbiting element (limited angle defined by _mid and _end)
	_end, _path	Pathed element

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

5.4 Areas

Areas are intended to change the direction of an element's progress or pause it. The only exception is the grip type that links the element progress to the movement of a person's hand. Areas are objects in the 3D model. Because their geometrical shape matters, Tames will need access to their vertex data.

IMPORTANT For accessing vertex data you need to let Unity access this data by checking the relevant option in the model's prefab inspector (See 8.2.1).

Areas are defined by their names. The first element of the name is their shape, followed by their interaction mode (the bold number of letters are mandatory):

*_**sha**pe_**i**nteraction…*

There are 3 types of shapes and 8 interaction modes, listed in Table 5.2. We can have any combination by one element from each column in Table x:

Table 5.2. Area markers (the bold parts are mandatory).

Underscore	Shape	Underscore	Interaction mode
_	**Cub**e or **box**	**_**	**P**ositive
	Sphere		**N**egative
	Cylinder		**I**nside
			Outside
			Grip

1-state switch (+ or *off*)

2-state switch (- or +)

3-state switch (-, *off* or +)

A revolving automatic door, for example, would need an area object named like ***_box_i**n*, while a sliding door would instead need something like ***_box_p**os*. The difference is that the first "**i**nside" mode activates the element only when a person enters the area, so the door only revolves when a person is inside the area. On the other hand, "**p**ositive" keyword always keeps the element activated but changes its progress direction based on if the person is inside or outside of it (if inside, the direction would be positive). So, the sliding door closes (decreasing progress value) when the person is outside and opens (increasing the progress value) if the person is inside.

IMPORTANT: the cylindrical shape needs to have at least **5** vertexes in its base with fan caps.

While the first four area types track the position of a person's head, the next four track their hands. The Grip type specifically tracks the gripping gesture and if the centre of the grip is inside the area's shape. The switch types, on the other hand, check if the tip of a middle or index finger has *just* entered the area's shape (so triggers the switch).

Areas can have another description in their names that determines how themselves would change (move or rotate). They can have a ***f**ixed* location, follow their ***m**oving* part of the element, or be ***l**ocal* to the element's base object. You don't need to define any of them in most circumstances because there is a default value for each (fixed for positive, negative, inside and outside types; attached to the mover for grip type; and attached to the elements for the switch types). However, if you wish to change them, you add an underscore character with the first letter of the changing types (**f**ixed, **m**over or **l**ocal):

*_**cub**e_**p**os_**f**ix… or _cub_p_f*

6 Basics of Tames manifests

Tames manifests are plain text files in which dynamic elements and some general settings are defined or elaborated. There are two manifest files already included in Tames template (Resources folder), called *manifest.txt* and *default.txt*. We recommend not to change the latter but you can modify the former as you wish and even include your own files. The structure of a manifest is similar to the lines in 4.1 and 4.2 (without comments in brackets), with a **case-insensitive** keyword that defines a property followed by its value(s). There are three types of keywords in Tames:

1. One-liner keywords (usually for general settings)
2. Block header keywords (the first line of an element's definition)
3. Content keywords (the properties and options of an elements)

6.1 General settings

Import manfest1, manifest2, ….

This line can only exist in the first line of the main manifest file. Tames will also load all other manifest files listed after the *import* keyword (separated by comma; spaces before and after the comma are ignored). You can use this feature to import interactive elements between projects without risking mistakes in each project's manifest.

Walk object1, object2, …

It defines the walkable surface by the listed objects. This line can be used multiple times anywhere in the manifest. Please see 7.1.1 for walkable surfaces and navigation

Eye value1,value2,…

This sets possible eye levels (distance from the camera over the closest walkable surface, or the Y-coordinate of the camera, if there is no walkable surface is defined). Values are floating numbers in Unity units. The values should be separated with commas **without** any space. The user can manually toggle the levels (see 8.3.1 for input controls).

Camera mode object1, object2, …

This attaches the position and/or orientation of the camera to different objects. The *mode* parameter can be *turn*, *move* or *both* which control what aspect of the camera is linked to the object. The user can toggle between the objects (and no object).

Mode index

This line sets the initial input mode by its index. The modes and their index include:

1. VR only: the user positions and orientates by moving and head rotation in real world.
2. VR headset + gamepad or keyboard: the user orientates by rotating head,but walks virtually by gamepad or keyboard.
3. Gamepad or keyboard: the user turns and orientates by gamepad or keyboard.

4. Keyboard + mouse: the user moves and turns left or right by keyboard but looks up or down by mouse.

6.2 Block headers

Object name1, name3, …

This line **declares** that the lines after it will describe the properties of the listed **existing** dynamic objects (separated by commas, but there can be a space after or before each comma). For additional listing methods please see 6.4. Please note that this line does **not** define 3D elements because they are automatically detected based on their markers.

Material name1, name2, …

This line defines that the listed existing materials (separated by commas) should be treated as dynamic. The listed materials **should be** already attached to objects within the *default* or *interactives* root objects. Otherwise, they will not be found.

Light name1, name2, …

This line lists the lights in *interactive* root object which we would like to treat as dynamic.

Custom name1

This line defines a new custom dynamic parameter. You can only define one parameter per block.

6.3 Block content

6.3.1 Updating elements

There are five ways that an element can be updated:

1. **Progress**: The progress of an element is updated based on the progress of another element. This is used for linking elements together. This type of update is available for all element types, and is set by the *update* keyword. There can only be one name after it:

   Update element-name

2. **Time**: It is same as the progress-based update, but the time passage is itself considered a (raw) progress. This is also possible for all element types and is default by Tames. So, you don't need to mention it in the manifest.
3. **Manual input**: This is also similar to the progress-based update, but the base progress value is changed by pressing keys, mouse buttons or other input devices. It is set by keyword *input*. It is possible to set an optional duration of the progress duration for the input progress. For example, a duration of 3 means it will take 3 seconds of holding the designated input key or button to move a full length of a progress. For the list of keys and pairs please see Table 6.1:

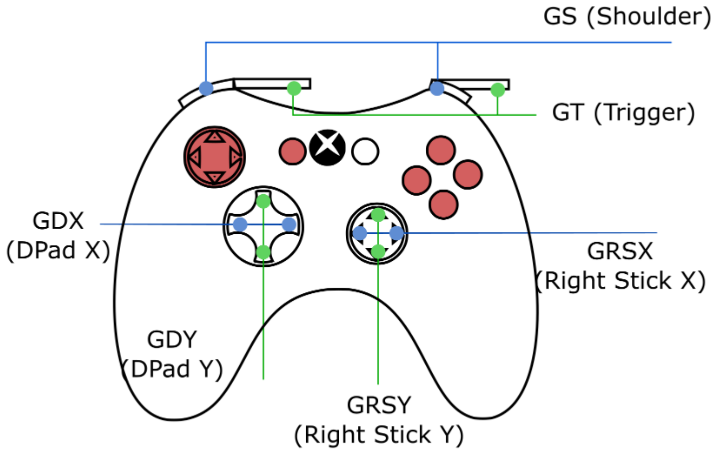
   Input [duration] [key-,key+] [pair]

4. **Position**: The progress of a 3D element is updated based on the position of an object. The moving part of the element moves to the closest possible position on its constrained path to the latter object. There are two possible targets of this movement: tracking an object (or person) or tracking the moving part of a 3D element. They are defined by *track* and *follow* keywords respectively.
   Please note that head and hand are **reserved** keywords in Tames and they will only be interpreted as head (camera position) and hands, and any 3D objects with these names will be ignored. You can have multiple objects listed after either keyword but only the closest to the element's mover will be considered in each frame. It is **IMPORTANT** to distinguish between the types of items listed after each keyword. After *track*, you list the name of 3D **objects** (or *head* or *hand*) but after *follow* you list the name of dynamic **elements**:

   Track head, hand, object1, object2, …
   Follow element1, element2, …

5. **Manual grip**: This is the same as tracking objects but with only hands in a grip gesture being tracked (more precisely the centre of grip). The grip update is defined by areas in the **3D model**, not in the manifest.

Table 6.1. Allowed controls for custom inputs in Tames

Device	**Control (Allowed / Not allowed)**
Keyboard	0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z Space Left Right Up Down Shift Alt Ctrl (only used with mouse button)
Game controller	 GS (Shoulder) GT (Trigger) GDX (DPad X) GRSX (Right Stick X) GDY (DPad Y) GRSY (Right Stick Y)
Mouse	Button

6.3.2 Progress management

The duration, speed, trigger and continuity mode of the progress can be defined in the manifest.

1. **Continuity modes** are defined by only their mode name (*stop*, *reverse* or *bounce*, and *cycle*). Only one continuity mode is accepted per block. In the absence of a keyword, Tames considers the mode as *stop*. To reduce number of lines in the manifest, it is possible to add the duration after the keyword:

   Cycle [duration]
   Stop [duration]
   Reverse/bounce [duration]

2. **Duration** and **speed** are defined by homonymous keywords. Only one of them may be used (the other is calculated as the reciprocal of the defined one), for example:

   Duration 2
   Speed 0.5

3. **Trigger** is defined by the sign of change direction after or before certain values of the update basis's progress of the element. Trigger is only applied if the element is updated by another progress (not by position or grip). To set it, after the keyword *trigger*, signs -. + or space are put before and after maximum two progress values (bother between 0 and 1, and the first value should be less than the second). Some examples are:

   Trigger -0.3 (negative direction under 0.3 and no change above it)
   Trigger +0.3 (positive direction under 0.3 and no change above it)
   Trigger 0.3- (negative direction above 0.3 and no change under it)
   Trigger -0.3+ (negative direction under 0.3 and positive above it)
   Trigger -0.3 0.7+ (negative direction under 0.3, positive above 0.7 and no change between)
   Trigger 0.3+0.7 (no change under 0.3 and above 0.7 but positive in between)

4. You can set the **initial** value of the progress to apply it when you launch the project:

   Initial [value]

5. **Area** keyword allows you to associate an element with areas of another element. This is useful for materials and lights, for which you cannot define an area in their 3D models. To use this keyword, you write the name of the element after it, and Tames will associate all areas for that element to this element as will (this does not work with **grip** areas), so the direction of this element's progress change is now determined by the presence in areas.

   Area element-name

6.3.3 Lights and material properties

Most properties of lights and materials follow the structure of *changers*. A changer sets the options and steps of changing a property. Their basic structure of a changer line, after the property keyword, is its changing mode followed by the list of steps, all separated by space:

[keyword] [mode] step1 step2 step3 …

The changing mode can be *grad*ual, *step*ped or a number representing an abrupt switch. The steps are associated with progress values, equally divided by the number of steps. However, if the switch mode is set, only the first and last steps are considered (Figure).

Some properties such as (keyword in brackets) albedo map's U and V offsets (*u, v*), emissive map's U and V offsets (*eu, ev*), and light's brightness or emissive intensity (*brightness*), emissive intensity are numerical, and so each step is only one floating number. However, others (*color* and *glow*) would change colours and so each step should be a colour. To define a colour step, you can either write the colour's name with a modifier (light or dark, see Figure x for the available colours) or use the RGB components of it (separated by comma). For example, the below lines are the same, setting a gradual colour changer for a material with three steps of light red, green and dark blue (note the **dash** in colour names):

Color grad light-red green dark-blue
Color grad 1,0.5,0.5 0,1,0 0,0,0.5

Each colour can have an additional component whose meaning depends on the type of element and keyword (Table). This additional component is written after a comma for each step. In the following example, the alpha channel for steps are set to 1, 0.5 and 0, respectively:

Color grad light-red,1 green,0.5 dark-blue,0
Color grad 1,0.5,0.5,1 0,1,0,0.5 0,0,0.5,0

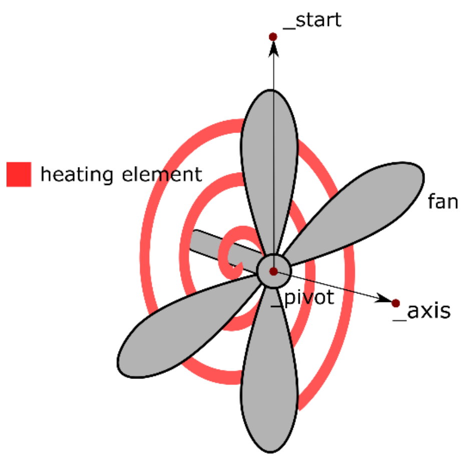
Element / **Keyword**	**Material**	**Light**
Color	Alpha channel (0-1)	Intensity (0-100000)
Glow	Emissive intensity (0-10)	Intensity (0-100000)

IMPORTANT When creating dynamic materials, a material with the same name on all objects is treated as one singular material. If you want to separate those materials and treat them separately, you should use a *unique* keyword in the content (no value needed).

6.3.4 Examples

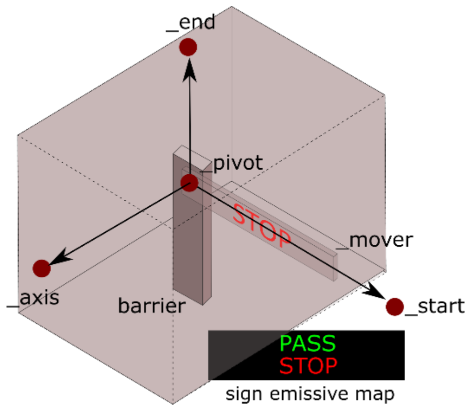
Lines below define a fan heater that turns on and glows red when temperature is less than 10°C (0.25 progress). The temperature is changed by pressing mouse buttons (using *input* keyword) It will take 3 seconds for the fan's heating element material to fully glow red:

```
Custom temperature
        Input 3 button
    Object fan
        Update temperature
        Trigger +0.25
        Cycle 0.2
    Material heating element
        Update temperature
        Glow gradual black red
        Trigger +0.25
        Duration 3
```



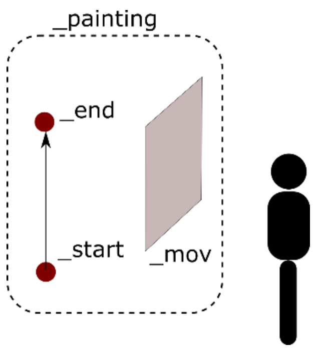
The following lines represent a barrier with a stop/pass lit sign, that is triggered by presence in an area. The sign's text (material's emissive V offset) would change at 0.9 of progress change (90% open)

```
Object barrier
    Duration 1
Material sign
    EV 0.9 0 0.5
    Update barrier
```



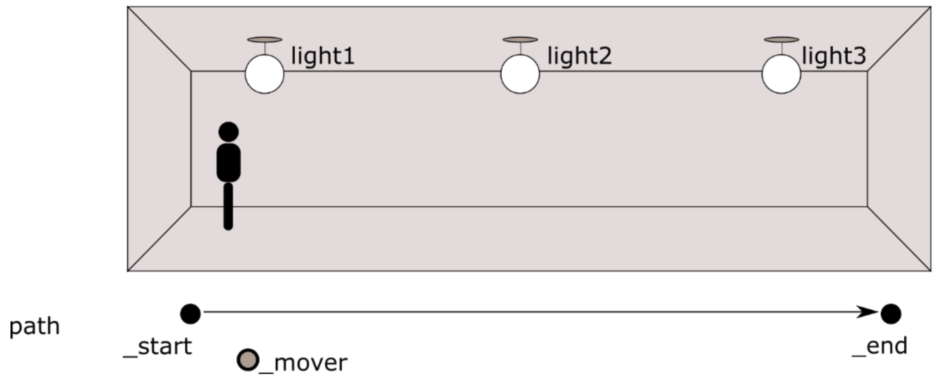
The following lines change the height of a painting to match the eye level of the nearest viewer. To ensure the painting's safety, it moves slowly (taking 3 seconds to slide one length of its path)

```
Object painting
    Track head
    Duration 3
```



These lines change the colour of lights in a rainbow order as the person moves right in the corridor. As the person moves, the path's progress changes and the lights' colour updates based on this progress.

```
Object path
        Track head
Light light1,light2,light3
        Update path
        Color gradual purple blue green yellow orange red
```



6.4 Working with names and arguments

6.4.1 Punctuation

As explained earlier, many keywords separate their arguments with spaces (though they may use commas in each argument to separate its components, e.g., in colours or inputs). The exception was keywords after which you need to list names. In this case, they were separated by commas instead. This is because it is more likely to contain a space in names, and so separation with space will negatively affect their reading. This also means that you would rather not contain an object with comma in its name.

In addition, you **should not** use spaces in the **beginning** and **end** of the object names, because for aesthetic reasons, Tames allows space before and after commas in argument lists (so it will not read spaces in names

6.4.2 Elements and objects with similar names

When using the name of an object or element, Tames will look only for the **first** object or element with that name and will ignore the rest. In other words, you cannot describe multiple homonymous objects. For example, imaging you have multiple elements named "door", none of the following lines will find them all:

```
        Object door
        Object door, door, door
```

To find multiple elements or objects with the same name or same starting partial name, we can use an asterisk (*):

*Object door**

This of course means that a *Door1* and *doorOfFridge* will also be included. So, please make sure you name your objects properly to remove the chance of inconsistency.

6.4.3 Relative name finding

Tames finds names based on a plain list of all objects and elements it has created on the launch of the project. So, there may be multiple objects or elements with same name sorted randomly. This means we can never be sure which name would be even the first that we would access. Therefore, we need to find our names based on their holders' relation with other objects.

For example, imagine we have multiple rooms in a house, but we only want to associate the door in one of them with an element. Considering all the doors have the same name (*door*), looking for a "door" will be useless. Tames allows us to look for the "door" of a specific room. In our case, our intended door may be the child of an object called *room1*. Therefore, we can find it as:

Object door<room1

In the line above, the sign < indicates that *door* is a child of *room1*. We are asking Tames to find an object named *door* whose parent's name is *room1*. We could have accessed *room1* with the reverse sign (find a *room1* whose has a child named *door*):

Object room1>door

If we want to find multiple doors in the same room or all rooms, we use the asterisk as in the respective lines below:

Object door<room1*
Object door<room**

Table 5B shows all signs used by Tames and their meanings. In all usages in the table, it is possible to add an asterisk after either or both sides of the sign. In this case, Tames will find all possible matches not just the first on.

Table 5B. Naming conventions and their signs

Sign	**Function: the first X …**
X<Y	… that is a child of the first Y
X>Y	… that has a child named Y
X=Y	… that has a sibling named Y
X<=Y	… whose father or uncle is named Y
X>=Y	… which has a child or nephew named Y
X<<Y	… that is a descendant of a Y
X>>Y	… that has a descendant called Y
X<<=Y	… that is a descendant of a Y or of one of its siblings

X>>=Y	… that has a descendant or grandnephew named Y

6.4.4 Element vs Object names

As mentioned, a 3D interactive or dynamic element is also a 3D object. Therefore, there is no inherent difference between the names of non-interactive 3D objects and interactive ones (3D elements). However, elements are identified automatically by Tames and are treated differently during the visualisation. This difference in treatment is not equal between the descriptive keywords in the manifest. Some keywords expect elements while other do not differentiate, or expect objects. It is ***IMPORTANT*** not to use normal 3D object names when element names are required and vice versa. In addition, some keywords only accept one element or object, and so listing multiple objects (by comma or asterisk) is not acceptable. Table 5C lists what each relevant keyword requires. Some of the keywords in this table are related to advanced 3D elements and are explained in the next section.

Table 5C. Requirements for each keyword

Keyword	Object or element	Multiple object
Walk	Object	Yes
Camera	Object	Yes
Object	Element	Yes
Alter	Object	Yes
Update	Element	No
Follow	Element	Yes
Track	Object	Yes
Link	Object	Yes
Clone	Object	Yes
Queue	Object	No
Area	Element	No
Scale	Object	Yes

7 Advanced 3D elements

7.1 People's movement

The movement of objects does not have any effect on the movement of people unless ther

7.1.1 Walkable surfaces

Walkable surfaces are normal objects that define the constraints of walking in the space. When they are imported, their upward-facing surfaces are detected and stored as the navigable area of the space. The movement of a walkable surface only affects the vertical movement of a person, if applicable. To influence the horizontal movement, please see the next section.

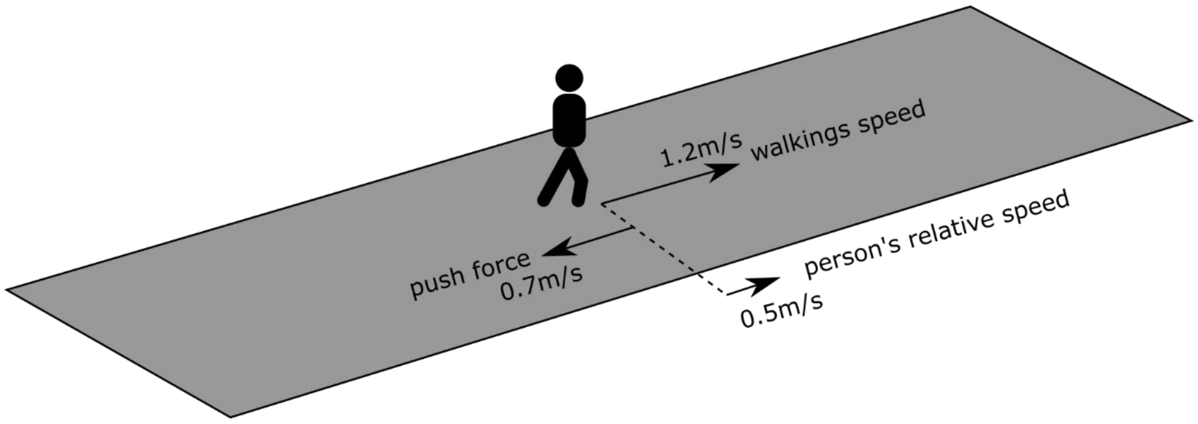
IMPORTANT Detecting model's surfaces requires Unity access to vertex data (See 8.2.1).

To set an object as a walkable surface you would need to add it after the "walk" keyword in the manifest (See 6.1). Without this line in the manifest, the entire space is walkable regardless the barriers or existence of a floor. The walkable surfaces will be visible throughout the navigation unless their name starts with an underscore (_).

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) in 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.

7.1.2 Push forces

A pushing force is a velocity vector that affects a person's default moving vector (Figure). Pushing forces are useful to simulate walkways an escalators.



Defining pushing forces works similar to defining moving objects, with only differences in the marker names (we add a "w" 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.

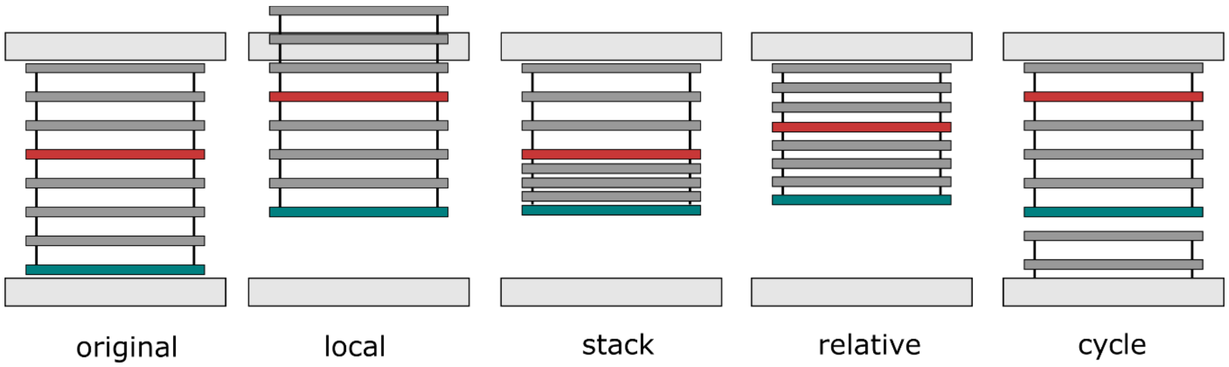
For sliding and pathed push, *_fstart* and *_fend* still indicate the direction of the push, but their distance represents the velocity (in m/s) or the force of the push. In rotating push forces, there is no start and end, but only *_fpivot* and *_faxis* that represent both the rotation axis and its 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).

7.1.3 Linked objects

Tames can link multiple objects to a parent object so they update accordingly. This is one of several ways to reduce the amount of markers in the 3D model. The *link* keyword in the object's content block is used as follows:

Link mode offset obj1,obj2,…

The linking mode can be *local*, relative, *stack* and *cycle* with their difference illustrated in Figure:



1. The *local* link mode imitates the movement of the parent object in the linked objects, local to their transform. For example, elements of a blind can all rotate individually based on one progress. We don't use the offset parameter in this mode.

   *Link local objects**

2. The relative mode moves the linked object relative to their position on the main object's path. The offset parameter is used to denote the
3. The stack mode creates a stack imitation of multiple objects based on the path of the main element. Tames imagines that all the objects have the same thickness and it calculates this value based on the number of the objects and the offset value. For example, for a 10-component 1m blind with each component being 2mm thick, the offset should at least be 0.02 (2cm = 10 x 2mm):

   *Link stack 0.02 bind-comp**

4. The *cycle* mode moves the linked objects in a cycle on the main element's path. The offset value is the percentile distance between the last and first objects in the cycle. This is used for when the movement path is not cyclic. For example, the following line can be used for simulating a sushi train.

   *Link cycle 0 plate**

7.1.4 Cloned elements

TBD

7.1.5 Queued elements

A queued element is an element with its moving part cloned along its path to form a chain or queue of moving objects. This is a shortcut for a linked object when all moving objects are the same (for example steps of an escalator). Similar to links, an offset value is necessary to set the distance between the last and first objects at the end of a cycle. Then we should decide if we want the clones replicated *by* a distance or in a certain *count*, followed by the value of either.

Queue offset by/count value

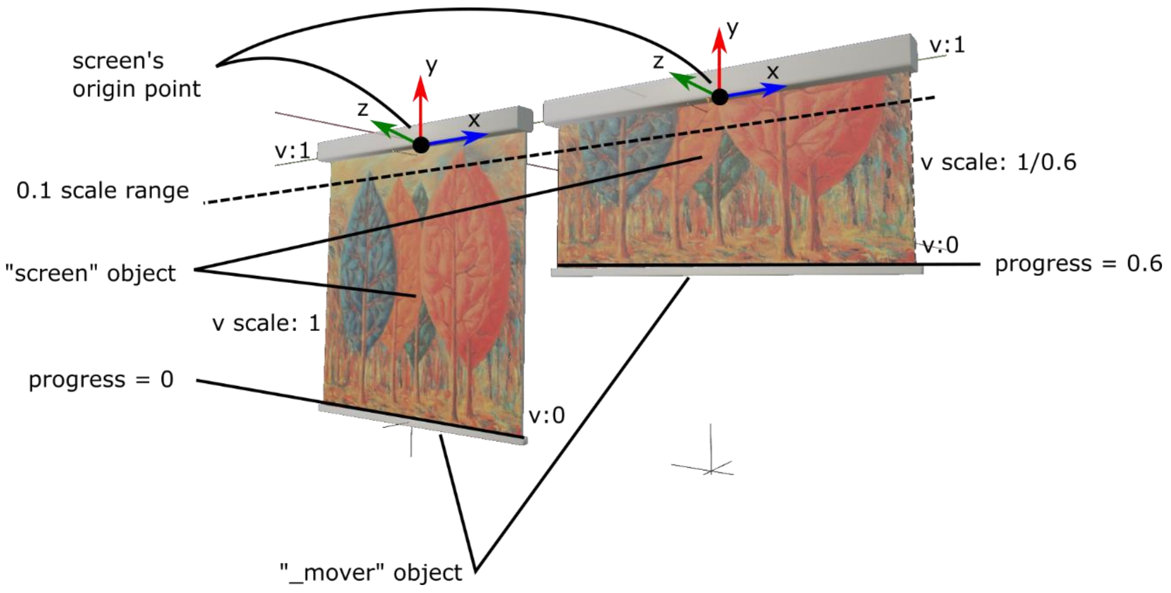
7.1.6 Scalable objects and texture

The keyword *scale* within an object's block content links to and scales another object. This is used for soft elements like ropes are sliding screens whose physical simulation of shrinking or rolling is difficult but not important because they are usually hidden behind a cover.

Scale axis from,to uv obj1,obj2,…

The first argument *axis* is *x, y* or *z* and it indicates which axis the objects are scaled by. Then the scaling range (*from* and *to*, separated by comma) are set, which proportionally correspond to the progress value (*from* to 0, and *to* to 1). The *uv* argument is x (or u) or y (or v) that sets the *uv* property of the material that scales so to retain the appearance of the object. For example the line below allows simulation of a screen like visual (Figure).

Scale y 0.1,1 v screen



IMPORTANT Using *_path* markers requires Unity's access to vertex data (See 6.3.1).

8 Unity

8.1 Installation and version management

Tames is developed using Unity 2020.3.33f, and therefore requires at least this version to work with. For installing a Unity version, you need to look for it in Unity Hub (you can download it from Unity website). You would also need a Unity account, that is free for students and personal use.

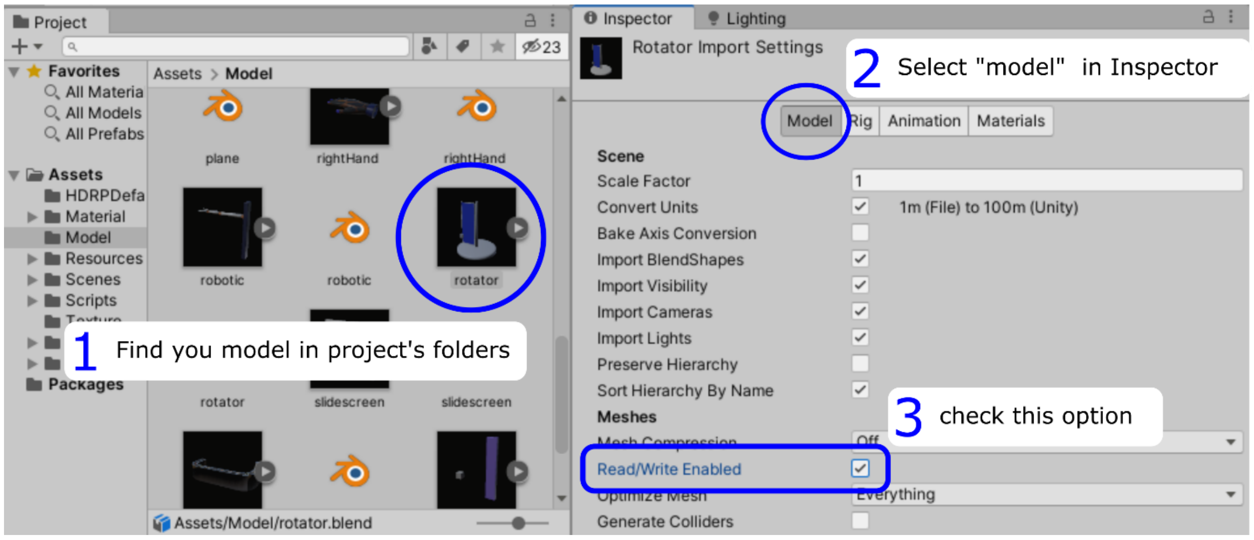
8.2 Importing files

You don't need to import Tames into Unity but only open a copy of it in Unity Hub. However, for importing your other files (models, materials, etc.) you only drag and drop them in the folder you like inside Unity Editor (the Project panel).

IMPORTANT When importing a file into a Unity project, Unity creates a copy of it. So, if your file is open when doing so, you need to close it and instead open the one you copied in your Unity project.

8.2.1 Read and write option

Enabling the read and write option is required for analysing the 3D data of a 3D model. Tames will need this data for several operations (but does not change any of the models). Please follow the instructions in Figure 8.1 to enable this option for each 3D file that contains an interactive object. Remember to click on Apply for each model after changes.



Scene
Scale Factor: 1
Convert Units: ✓ 1m (File) to 100m (Unity)
Bake Axis Conversion: ☐
Import BlendShapes: ✓
Import Visibility: ✓
Import Cameras: ✓
Import Lights: ✓
Preserve Hierarchy: ☐
Sort Hierarchy By Name: ✓
Meshes
Mesh Compression: Off
Read/Write Enabled: ✓
Optimize Mesh: Everything
Generate Colliders: ☐

Figure 8.1.

8.3 Launching and playing

To launch a project, you press the Play button around the top middle of your Unity window. Please be patient as it may take a few seconds for Unity to start the project. If you press the Play button during this, Unity assumes that you have pressed Stop (as the button changes to Stop during play mode).

8.3.1 Reserved inputs

Some keys and buttons are reserved for operations during the project's play mode, and cannot be changed or assigned to input elements in the manifest. Figure 8.2 shows the reserved controls (their active status depends on the active control mode):

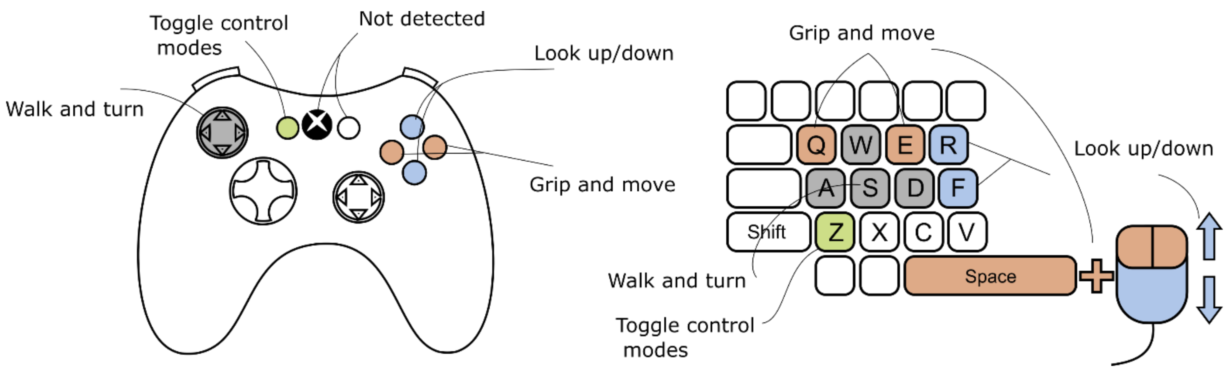


Figure 8.2. Reserved controls for Tames