Boku Tutorial Support

This document is a description of systems that Boku’s software tutorials support today.

The current system is incomplete but has some basic functionality. There is enough done to show that the direction of the current design is worth continuing.

# Class Descriptions

The following are the primary classes a tutorial writer has to deal with today. Although thoughts of putting more of this into a more designer friendly format other than code continues to be a design goal, there are several primary implementation details that directly control or monitor game state that are far easier to manage in code.

## Class Tutorial

This represents the base class to create the tutorial from. It has several members that are important to understand and use.

### protected List<Instruction> instructions

This instructions member is the list of instructions to step through. The tutorial writer will create these and add them to the list. Currently today this is a flat list and not a graph; which has some limitations but with our goal to provide simple tutorials this is not a big restriction. See Class Instruction below for more details.

### protected Boku.Input.InputConstraint inputConstraint

This inputConstraint member describes the active input constraint to apply to the input system. This allows the tutorial writer to focus and restrict the user to the set of actions the tutorial can handle and thus help limit the complexity of state handling to direct the user. See Class InputConstraint below for more details.

## Class Instruction

This represents the base class for a single instruction. Only one instruction will be active at a time within the tutorial. As of today, all instructions are text only (no icons or graphics).

### Display Properties

The display properties control the format and content of the instruction.

#### public string title

This is the text that represents the title of the instruction; often an abbreviated single line that describes the instruction.

#### public string text;

This is the text the represents the body of the text of the instruction; often a short paragraph with details that user will need.

#### public Alignment alignment

This is the alignment of the instruction itself. This allows the instruction to be “placed” at different parts of the screen. The default today is top right.

### Control properties

The control properties will define the running behavior of the instruction.

#### public Modality modality

This defines how the instruction affects the users control when it is active. The three possible values are Modeless, Modal, and ModalWithContinue.

Modeless means it does not affect the users control.

Modal means that it will block all user input until the instruction is no longer active

ModalWithContinue means that it will block all user input until the user uses the standard instruction continue input (currently the A button on the game controller).

#### public float durationSeconds

This defines how long the instruction should be displayed and be active to the user. Normally only used when the property modality is set to Modal. Once the timer period has expired, this instruction will be deactivated and the next one will be activated.

#### public List<Condition> conditions

This defines the set of conditions that must be met to before this instruction is considered completed. A Condition is a delegate that is called to provide the specific state.

Often a condition consists of a UI mode (see Class App)

#### public List<Hint> hints

This defines the set of hints for this instruction. The Class Hint is described below. While the instruction is active, the hints provide context relative text that will help direct the user to do actions that will complete the instruction. See Class Hints below for more details.

#### public Modify Starting

This allows the Tutorial to add an event that is called when this instruction is first starting. This is a helpful place to customize the InputConstraints for this instruction.

#### public Modify Finishing

This allows the Tutorial to add an event that is called when this instruction is about to stop. This is a helpful place to reset or clear customization made to the InputConstraints for this instruction.

## Class Hint

The represents a “hint” that is added to an instruction. It is often created and used directly and rare to derive a custom class from. Although multiple hints can be active at once, the current display system does not manage them so they are not overlaid. As of today, all hints are text only (no icons or graphics).

### Display Properties

The display properties control the format and content of the hint.

#### public string title

This is the text that represents the text of the hint .

### Control Properties

The control properties will define the running behavior of the hint.

#### public float durationSeconds

This defines how long the hint should be displayed and be active to the user.

#### public float hintRepeatSeconds

This defines how long to wait after the conditions have been met the first time before the hint is shown. It also defines how much time to wait to repeat the hint displayed if the conditions are still met and it is no longer displayed.

#### public bool repeat

This activates the ability for the hint to be repeated after its first use.

#### public List<Condition> conditions

This defines the set of conditions that must be met before this hint can be activated and displayed. A Condition is a delegate that is called to provide the specific state.

Often a condition consists of a UI mode (see Class App)

#### public Modify Starting

This allows the Tutorial to add an event that is called when this hint is first starting. Not common to use for hints.

#### public Modify Finishing

This allows the Tutorial to add an event that is called when this hint is about to stop. Not common to use for hints.

## Class InputConstraint

This class is used to manage the active input constraints for the input system. It is primarily used by the Tutorial (a member of one) to expose the management. The issue today is that not all of Boku’s user interaction uses one system nor do all systems allow this to be used without interjecting specific checks at every use location scattering lots of code and definition throughout the code. This should be improved.

### public RestrictionTypes restriction;

This member allows the Tutorial to define how the exceptions list is used. This allows the tutorial to define the smaller set of items, the disabled set or the enabled set.

EnableAllExcept will use the exception list as a set of items to disable.  
DisableAllExcept will use the exception list as a set of items to enable.

#### public List<string> exceptions = new List<string>();

This is the list of identifiers for the input exceptions. Currently these are strings to allow the most flexibility (late binding) of creating tutorials or UI input. It could be replaced with an enum that included an entry for every input description (like NavigateMenuUp, not StickUp). See Input classes like CommandMap for more details how to define these when using that system.

## Class App

This class wraps and exposes application information in a very easy Tutorial interface. Today it only exposes a single property.

#### public string UiMode

This property returns a string that describes the current UI mode. All UI modes are unique. These are defined by the CommandMap id; either from the dummy command map created in code or the command maps loaded from the input XML files. In some cases due to different Input systems, it is required to call the Sim.UiMode to get the correct mode. See class Sim below.

## Class Sim

This class wraps and exposes simulation information and control in a very easy Tutorial interface. Today it has only a small set of features that are active.

#### public string UiMode

This property returns a string that describes the current UI mode in the sim.

#### public Thing FindThingByClass(string idClassThing)

This method will find and return a Thing that represents the class thing. Exact class name is not needed.

#### public Thing FindThingById(string idThing )

This method will fine and return a Thing that represents the instance of a GameThing. The id of the instance can only be set today by modifying the saved game’s stuff file and adding the id property.

#### public Thing AddThing(Vector2 position, string idClassThing)

This method is not implemented yet. It will allow the tutorial to add things at specific locations to allow tutorials to expand the set of things to interact with as it progresses.

#### public void Pause()

This method will cause the sim to go into a paused mode.

#### public void Play()

This method will cause the sim to go into a running mode.

#### public Camera camera;

Although partially implemented, this class currently doesn’t do anything. It is meant to allow the Tutorial to change the view to better suit the tutorial.

#### public List<string> thingsAvailable;

This is the set of identifiers for the things the user can see in the Selector that then can be placed into the world. Currently not implemented.

#### public List<string> thingsDisabled;

This is a set of identifiers for the things that will be disabled in the Selector that then can not be placed into the world. Currently not implemented.

# Creating and Attaching a Tutorial Class to a level file

See TestTutorialA.cs for an example code; but I have included snippets below.

First, the tutorial writer should create a class derived from Tutorial that has a unique name from other tutorials. Currently these should be placed under the Tutorial\Tutorials\ project branch.

Then, the tutorial write should provide a constructor for their class in which they define the instructions, hints, and input constraints for their tutorial.

Lastly, they should create a tutorial level file and save it. Then within that XML file, they should associate it with their tutorial class by adding the XML tag <tutorial> with the contents being the class name. See Content\Xml\Levels\Missions\TutorialTest.Xml for an example.

## Setting Input Constraints

To change the input constraints, just modify the tutorial member “inputConstraint”. When the tutorial starts it will be used during the life of the tutorial. Instructions can modify it when they become active by adding a delegate to the Starting and Finishing events on the instruction.

this.inputConstraint.restriction = Boku.Input.InputConstraint.RestrictionTypes.DisableAllExcept;

instruction = new Instruction();

instruction.title = "Tutorial on using tutorials";

instruction.text = "This tutorial will walk you through using a tutorial.\n";  
instruction.Starting += delegate()   
 {

this.inputConstraint.exceptions.Add( "SaveMenu.NavPrev" );

this.inputConstraint.exceptions.Add( "SaveMenu.NavNext" );

this.inputConstraint.exceptions.Add( "SaveMenu.Select" );

this.inputConstraint.exceptions.Add( "Sim.SaveMenu" );

};

instruction.Finishing += delegate()

{

this.inputConstraint.exceptions.Clear();  
 this.inputConstraint.restriction =   
 Boku.Input.InputConstraint.RestrictionTypes.EnableAllExcept;

};  
instruction.modality = Instruction.Modality.ModalWithContinue;  
this.instructions.Add(instruction);

## Implementing Instructions

Within the constructor for the Tutorial, you just need to create an instruction instance, set its properties, and then add it to the instructions. The order of the instructions is the order that they are ran in.

// first instruction

instruction = new Instruction();

instruction.title = "Tutorial on using tutorials";

instruction.text = "This tutorial will walk you through using a tutorial.\n";

instruction.text += "It will have all the intructions you need to accomplish using a tutorial.\n";

instruction.text += "When you are ready to continue, you can press the A button on the gamepad.";

instruction.modality = Instruction.Modality.ModalWithContinue;

this.instructions.Add(instruction);

// next instruction

instruction = new Instruction();

instruction.title = "We control the Vertical, we control the horizontal";

instruction.text = "From time to time we will need to take full control. ";

instruction.text += "Don't worry, will give you back some control as you progress; but right now you will notice that you can't do anything using the gamepad.\n";

instruction.text += "Go ahead and try, we will wait...";

instruction.modality = Instruction.Modality.Modal;

instruction.durationSeconds = 30.0f;

this.instructions.Add(instruction);

## Implementing Hints

Within the constructor for the tutorial, as you create each instruction, you just need to create a hint instance, set its properties, and then add it the instruction.hints.

// next instruction

instruction = new Instruction();

instruction.title = "Lets do something real usefull";

instruction.text = "Now I am going to teach you how to exit a running game (or tutorial) and return to the main menu. ";

instruction.text += "You can this by pressing the Back button on the gamepad and select the Exit to Main Menu option.";

instruction.modality = Instruction.Modality.Modeless;

instruction.conditions.Add(delegate() { return App.Instance.UiMode == "App.TitleMenu"; });

instruction.Starting += delegate()

{

this.inputConstraint.exceptions.Add( "SaveMenu.NavPrev" );

this.inputConstraint.exceptions.Add( "SaveMenu.NavNext" );

this.inputConstraint.exceptions.Add( "SaveMenu.Select" );

this.inputConstraint.exceptions.Add( "Sim.SaveMenu" );

};

instruction.Finishing += delegate()

{

this.inputConstraint.exceptions.Clear();

this.inputConstraint.restriction = Boku.Input.InputConstraint.RestrictionTypes.EnableAllExcept;

};

// add a hint

hint = new Hint();

hint.title = "Please press the back button on the game pad to get to the Save/Exit menu";

hint.conditions.Add(delegate() { return Sim.Instance.UiMode == "RunSim" && App.Instance.UiMode != "App.MiniHub"; });

hint.durationSeconds = 10.0f;

hint.hintRepeatSeconds = 15.0f;

instruction.hints.Add(hint);

// add a hint

hint = new Hint();

hint.title = "Please grab the gamepad.";

hint.title += "The back button is the small button to the left of the XBox button.";

hint.conditions.Add(delegate() { return Sim.Instance.UiMode == "RunSim" && App.Instance.UiMode != "App.MiniHub"; });

hint.durationSeconds = 10.0f;

hint.hintRepeatSeconds = 40.0f;

instruction.hints.Add(hint);