Setting up an Experiment with the Unity® Engine

Abstract

Dear all, this tutorial is about how to implement an experiment with the Unity® engine. Three hours are just not enough time to cover the vast amount of features of the engine. But I hope we can have a look at some of the crucial features. In the first section, you will find some general information regarding the installation. After this, a short overview of the engine is given. Then I describe the intended scenario as well as some of the covered points. At the end you will find some tutorial suggestions. I hope the tutorial covers the points you are interested in, please let me know if you miss something and I will do my best to incorporate the respective issue in the tutorial.

Contents

1 Installing Unity and the Oculus Runtime

The tutorial will rely on the free version of Unity®, features of the pro version are not covered. Installers are available for Mac OS X (10.8. and above) and Windows (at least Windows 7 SP1). A detailed description of the system requirements can be found here. Installing Unity on Linux systems is possible, however, it requires a Windows runtime environment, like Wine. A description for the setup can found in the Unity Community Wiki. By now, there is also an experimental build, but it relies on a - comparatively - dated version of Unity® with limited VR support.

During the installation, you have the option to install Visual Studio Community 2015 as the default editor. If you have no other C# editor installed, than I would recommend you to install it. However, this can take pretty long. In the end you can edit the scripts with any editor you like, Unity® will compile the code.

For the tutorial, it is not necessary to install the Oculus runtime, but if you would like to test the provided code with your own HMD, than you can get the runtime from the download section. There you can also find the different engine integrations, for instance for Unity® 5. I tested the samples with version 0.8 of the runtime, which you can obtain here.

2 The Unity® Engine

Unity® is a game engine which provides 3D rendering capabilities (2D as well) in combination with a physics engine[1]. Further, Unity® comes with a powerful animation system, called mecanim (which is not covered here). Unity® provides interfaces for quite some VR related devices. Most HMDs are supported (Oculus, Vive, etc.), as well as some common motion capture systems (Leap Motion, Kinect, but also Vicon systems). One of the main advantages of Unity® is the ability to deploy binaries to various platforms - you can build your projects as mobile apps, browser applications, or usual programs (Windows, Mac OS, Linux systems). Further, there is a vast and very responsive community. On the downside, Unity® does not support the usage of powerful GPU functionality for rendering and physic simulation (if it would, you would lose the mentioned portability). If your project requires really high-end graphics and involves complex physics, the Unreal engine will probably the better choice.

The Unity® distribution you installed provides the Unity® editor, a graphical user interface that allows the generation of scenes, which can be exported as applications. While the editor allows to create objects, define animations, edit physic settings, or adjust the global lighting, scenes become active due to scripts.

[1] A lightweight version of NVIDIAs PhysX engine.

The scripts allow to respond to user input, to generate objects on the fly or to initiate events under certain conditions, for instance if two objects collide. Unity® supports the usage of different scripting languages, namely Boo (a python like language), JavaScript and C#. This tutorial focus on C# since it is the most powerful of the three possible languages. Since C# compiles into the Common Intermediate Language (CIL, the assembly language of the .NET and Mono frameworks), it has access to quite low-level system functionality. For instance you can easily import windows system libraries (this is not possible with Boo or JavaScript). The languages can be mixed in one project.

Even if the editor is very powerful, you will likely have to do some programming when working with Unity®. The API documentation is very detailed, and features a lot of examples. Further, there are a lot of video tutorials available, which are organized in different topics, like VR. Further, the community is very responsive and supportive, most answers in the forums contain sample code.

3 Target Scenario

VR setups allow to immerse participants into standardized, yet realistic worlds, which hopefully allows ecologically and internally valid experiments, with manipulations which are not feasible in real-world or classic lab setups. The sample setup will be somewhat similar to the walkway navigation tasks used by Sprague and Ballard (2007) to investigate visuo-motor control in complex environments. Here, participants had to walk on a virtual path thereby avoiding obstacles and collecting target objects. We will have a look how to implement such a setup in Unity® and how to run it on the Oculus.

3.1 Object Creation and Physics

First, we will have a look at basic object creation and some aspects of the physics engine:

- Generate and place geometrical primitives
- Enable physical behavior
- Have a closer look at the visual and physical update cycles and the respective temporal uncertainty
- Check how to obtain diagnostic information like the frame rate, CPU load and the like

3.2 Respond to Input

Second, we will have a look at Unity®'s input system. Even if we will rely on the optical tracking of the Oculus for locomotion and the inertial tracking for the head orientation, other input systems are useful to test and debug the experiment before deployment:

- Locomotion via keyboard input
- Rotation with the mouse
- Custom input mappings

3.3 Keeping Track of the Actors Position

Third, we will examine how to track the users position and head orientation in the scene. To standardize the progress of the trials, it is necessary to assure comparable initial conditions - like the starting position and the gaze direction. Further, throughout a trial it is necessary to monitor whether the participant left the intended path:

- Continuously check whether the actor is within a certain bounding volume
- Check if the user is in a start position, and provide a visual cue otherwise
- Simple "fixation" check via ray casting

3.4 Object Movement and Object Presentation

Fourth, we will generate some obstacles which the actor has to avoid while moving. Since we want a challenging environment, we will make the obstacles only visible, when the actor approaches:

- Setup objects and assign movement patterns
- Create obstacles that approach the actor
- Implement dynamic fading which hides or shows obstacles depending on the distance of the actor

3.5 Speech Recognition

Fifth, we will integrate simple language recognition using the Microsoft Speech API. Since our participants have to walk around, input via keyboard or mouse is not feasible (and not natural), a controller with a long cable might be a solution,

but carrying it around can be cumbersome. Hence we will use spoken language as response modality:

- Access external programs via a local TCP/IP interface
- Process speech input

3.6 Setting Up the Controller

Now we have everything that is necessary to assemble the experiment: We can move through the scene, we can standardize the initial conditions, we can collect responses, and we have the obstacles and targets. To assemble this into an experiment we need a controller that handles the execution of the whole experiment and the single trials. There are different design patterns that can be used, we will model our experiment in terms of a finite-state machine. We break down the experiment in several stages, like instruction, training and so on and implement the respective functionality and the transitions:

- Define the overall structure of the experiment
- Handle data collection and serialization, i.e. writing data to a file
- Control trial progression

3.7 VR Support

Until now we were working without any VR specific tools. In the next step we will integrate the Oculus and deploy the experiment as an executable program:

- Enable VR support
- Configure the initialization of the HMD
- Build a binary

3.8 Controlling Application via Command Line Arguments

Now we have a VR-enabled program, however, we cannot configure it anymore via the editor. In the last part of the tutorial we will have a look how to forward parameters to the binary to customize the experiment:

- Using command line parameters via batch files
- Setting up a start-up dialog in C#

This is the general outline, I hope the points you are interested in are covered. If you miss a certain point than just write me before the tutorial and I will try to incorporate it.

4 Further Tutorial Suggestions

As mentioned above, Unity® supports different scripting languages, I would suggest to use C#. There is a dedicated section on the tutorial page, which is concerned with scripting. Most tutorials feature a video guide and code samples.

If you are interested in the basics, I would recommend the following tutorials:

- Scripts as Behaviour Components: This tutorial introduces the interface between the editor and scripts that is how to assign scripts to objects via the editor.
- Awake and Start: All scripts that are assigned to objects are derived from the MonoBehaviour class. This class defines various events, like Awake and Start which can be used to initialize the respective scripts.
- Update and FixedUpdate: Two other important events are Update and FixedUpdate, they are called once per update cycle of the graphics (Update) and the physics engine (FixedUpdate).
- Instantiate: In object oriented languages (like C#) you are usually using so called constructors to create objects. In Unity® you will usually rely on the Instantiate method.
- Activating GameObjects: You can enable and disable objects at runtime, this is quite useful to avoid heavy CPU load due to complex operations that not need to be carried out all the time (for instance our fixation check should only run at certain times). Activating / Deactivating is one possible solution in such cases.

C# in general is a versatile programming language, some of its features are covered in the intermediate section, I would recommend the following tutorials:

- Lists and Dictionaries: These data types are more flexible than simple arrays, especially if you need to create some kinds of mappings, dictionaries are the data structure of choice.
- Coroutines: Sometimes it is necessary to handle tasks in parallel, or in terms of background tasks. Due to its implementation, it is not advisable to use the common C# threading system in Unity®. If you want to parallelize tasks, Coroutines are the method of choice.

- Delegates: Sometimes you want to respond to events with the invocation of a certain function. For instance, you might want to inform the main script if the participant walks into an object. Delegates are an elegant way to realize such behaviors.

All these tutorials are detailed and provide example code. However, they focus quite exclusively on the respective concept. If you would like to see these concepts mixed and applied in a broader context, I would recommend the following tutorials:

- A 3D Fractal: This tutorial features dynamic object generation at runtime and shows how to manipulate object properties on the fly.
- Swirly-Pipe: A small racing game that covers a lot of the engines features.

In contrast to the tutorials on the Unity® page, the catlikecoding tutorials feature the complete setup of a project.