Cubiquity For Unity3D Quick Start

Introduction

This document provides a quick overview of Cubiquity for Unity3D. Documentation is rather thin at the moment because the system is developing rapidly, but hopefully this will help get you started. We would also strongly recommend that you watch our introductory video on YouTube as that will introduce you to the basics of the system

//Video

Creating Volumes

The fundamental building block of Cubiquity is the volume, which is a collection of millions (or potentially billions) of voxels. In the current system each voxel is simply a colored cube, though other representation (e.g. for smooth textured terrain) are coming in the future. Volumes can be created using the Unity editor or through code.

Through the editor

You will find the volume creation wizards in the main menu under 'GameObject' -> 'Create Other' -> 'Colored Cubes Volume' and then choose the appropriate wizard (they are all demonstrated in the video). Volume data is stored as a folder rather than as a single file, so please create an empty folder for you data in 'Assets/StreamingAssets/Cubiquity/Volumes'. The wizard will enforce this location.

Through code

A voxel terrain consists of a Unity GameObject with a ColoredCubesVolume component attached. To create such a pairing you must use the ColoredCubesVolumeFactory. Note that you cannot simply create a ColoredCubesVolume component using new and attach it to you own game object. In this sense it works the same as Unity's own built in terrain, in that this lets you create a GameObject with a terrain already attached but it doesn't let you attach a terrain component to your own object.

You can see example code by looking at the volume creation wizards, such as 'CreateEmptyColoredCubesVolumeWizard'.

Modifying volumes

Real-time modification is one of the distinguishing features of voxel-based environments, and again this functionality is exposed through both the editor and through code.

Through the editor

Current editor functionality is very basic. When you select a volume in the hierarchy view you see its inspector appear. This provides the option to add, delete, or paint voxels by clicking on the volume, as well as changing their color. This ability to draw individual voxels into the volume is basic but important, and in the future we will build higher-level functionality (lines, fill, etc) on top of this.

Through code

The volume has a very simple interface, which simply allows you to read or write the color of individual voxels. First you need to get a reference to the ColoredCubesVolume component, and then you can call its GetVoxel() and SetVoxel() methods. Whenever you modify a voxel the rendered image of the volume is automatically updated on the next rendered frame. Examples of this can be found in ‘ClickToDestroy.cs’, ‘CreateProceduralColoredCubesVolumeWizard’, and ‘ColoredCubesVolumeEditor’.

Creating a standalone build

You can create stand alone builds which use Cubiquity but there are a couple of things to be aware of.

1. Cubiquity projects rely on the Cubiquity native code DLL, which you will need to manually copy next to your generated executable. After you have made a standalone build you will find ‘CubiquityC.dll’ in ‘StreamingAssets/Cubiquity’, and you should copy this so that it is in the same folder as the executable. If you do not do this then it will automatically get copied the first time you run the standalone build, but unfortunately this copy happens too late for the library to be loaded by Unity (it will then load successfully on successive executions). We will look for a better solution to this in the future.
2. Make sure that your volumes are placed in the ‘StreamingAssets/Cubiquity/Volumes’ folder, otherwise they will not load successfully. The volume createion wizards should enforce this anyway.