Duplicating, instancing, and sharing of volume data

The contents of this section are subject to change in the future. We are still working out what the technical and user interface issues are with regards to multiple volumes.

General considerations

Cubiquity for Unity3D provides full support for having multiple volumes in a scene. However, these are two main things to keep in mind and which we shall state here at the beginning.

1. We believe there are fairly specific scenarios which are appropriate for multiple volumes, and in many cases you will be better off having a single volume in your scene. We describe some example scenarios later in this document and explain how they may best be solved.
2. If you do work with multiple volumes, you should avoid having multiple volumes referencing the same VolumeData instance, and you should *probably* avoid having multiple VolumeData instance referencing the same voxel database. In particular, the use of Unity's 'duplicate' functionality can lead to problems if applied to Volumes and/or VolumeData assets.

These are not hard-and-fast rules and there are exceptions. The purpose of this section is to help you understand the issues which can occur and to identify which solution is right for you.

Note

We try and keep track of occasions when a user violates the rules and issue a warning if appropriate. Tracking this behaviour is not easy in editor mode because object lifetimes can be hard to predict. If you feel that you are being incorrectly warned about multiple usages then you may need to reload the scene and/or restart Unity.

Rules for voxel databases

A voxel database (.vdb) is the file which holds the actual voxel data which will be used by the volumes. It is a Cubiquity-specific format (internally built on SQLite) which Unity cannot directly understand. Duplicating these .vdb files within Unity appears to simply make a copy of them, which is a safe operation. After duplication, and changes will only affect the given .vdb and not the one from which it was duplicated.

Rules for VolumeData instances and assets.

The VolumeData class is just a thin wrapper around a voxel database, and the most important piece of information it stores is the path to the voxel database (a .vdb file). If you clone or duplicate a VolumeData asset using the built in Unity functionality then the new VolumeData instance will store the same path. Unity does not give us a chance to intercept, override, or modify this behaviour.

The situation is unfortunate, because if two VolumeData instances are pointing at the same voxel database and are modified through code or the Unity editor, then there will be a conflict when attempting to save the changes. There is no reliable way to resolve this, so in most cases you should

avoid duplicating or cloning VolumeData assets and instances.

The exception to this rule is if you set the write permissions of the VolumeData to be read-only. This can be done either through code by setting the VolumeData.writePermissions member, or it can be done in the editor using the inspector. In this case the underlying voxel database will be opened in read-only mode and Cubiquity for Unity3D will not attempt to save changes.

Note that you can still modify a read-only volume data (e.g. through GetVoxel()/SetVoxel()) but the modifications will be lost when the instance is disabled or destroyed. You can even have multiple VolumeData instances accessing a read-only voxel database, and any (temporary) changes you make are specific to the given VolumeData instance. Again, such changes are later lost because of the read-only nature of the voxel database.

Rules for Volume instances

Volumes have a VolumeData instance attached and through this they access the underlying voxel database. If a volume is duplicated then Unity appears to make the duplicated volume share a VolumeData instance with the original volume. This means that any changes to one volume are immediatly visible on the other volume. We do no know of a scenario where this would be desirable, **so we strongly suggest that you avoid duplicating volume instances.**

Scenarios

We now look at a few possible scenarios which exemplify the way in which multiple volumes should or should not be used.

Modeling a large terrain

It may seem tempting to break a large terrain down into smaller volumes in order to benefit from optimizations such as occlusion culling. However, Cubiquity already implements such optimizations internally and attempting to use multipl volumes in this scenario will likely lead to a loss or performance.

Modeling a city

If you have a city with a number of identical buildings then it might seem desirable to represent the building as a read-only volume and then place multiple instances of it. However, it is much better to have a single volume representing your whole world, and to write the voxel data for the buildings directly into this single main volume in multiple locations.

As will as giving better performance, this will make it easier to update the volume in response to events such as explosions. With multiple volumes you would need to iterate over then to determine which one are affected by the explosion, but with a single volume this becomes trivial.

The downside of this is that it may be harder to build such a volume, as you need a world-creation process which can build much larger environments.

Modeling planets/asteroids

This is a scenario where we believe the use of multiple volumes is appropriate, and an example is included with Cubiquity for Unity3D (the 'Solar System' example). In this case we provide a single voxel database which is referenced by two VolumeData assets, both of which have their read-only flags set. These two VolumeData assets are used by two seperate volumes representing the Earth and the Moon.

The voxel data is actually a series of concentric spheres with different material identifiers, representing the layers which are present in a typical planet. The actual material and set of textures we apply can differ between the volumes, giving the Earch and moon different visual appearances. In play mode it is possible to independantly modify each of the bodies as the changes are stored in a temporary location, but as mentioned previously these changes cannot be written back to the voxel database because it is opened as read-only.

This scenario is appropriate for multiple volumes primarily because we want to have different transforms applied to each (i.e. the Moon orbits the Earth while the Earth orbits the Sun).