Main Principles

We have worked hard to make both the visual and the programmatic interface to Cubiquity for Unity3D as simple and intuitive as possible. However, internally it is a powerful and flexible system, and with this comes a certain amount of complexity. A high-level understanding of how the system is works will help you to use it more effectively, particularly when using it from code.

Therefore we use this part of the user manual to provide the information which may be relevant to people who wish to push the system as far as possible, or even just those who like some understanding of what is happening behind-the-scenes. We will talk at a high level about the structure of the system, the algorithms which underpin it, and the design decisions we have made for its implementation.

Before proceeding to read about these principles we would recommend that you at least work through the **Quick Start** guide to get a feeling for what the system can do. This will provide you with some useful context when reading about the underlying details.

'Cubiquity' vs. 'Cubiquity for Unity3D'

One of the first points to understand is the difference between 'Cubiquity' and 'Cubiquity for Unity3D'. The first of these, 'Cubiquity', is a native code (i.e. C/C++) library for storing, editing, and rendering voxel worlds. It is not tied to any particular game engine or platform, and can be used from a variety of languages. On the other hand, 'Cubiquity for Unity3D' is a set of C# scripts which connect Cubiquity to the Unity3D game engine. These scripts allow Unity3D games to create, edit and display Cubiquity volumes. As a user of Cubiquity for Unity3D you do not (normally) have access to the underlying C/C++ code of Cubiquity, but you do have a compiled version of the Cubiquity library and the source code to the C# integration scripts.

Voxel Engine Concepts

Cubiquity is a *voxel engine*, which means that it represents it's objects as samples on a 3D grid. This is conceptually similar to the way a bitmap image represents a photograph as samples on a 2D grid (the pixels). In many ways a voxel can be considered the 3D equivalent of a pixel. The value stored at a voxel may simply be a colour (as in the case in our ColoredCubesVolume) or it may be some more advanced representation.

Rendering such a 3D grid directly is not trivial on modern graphics cards as they are designed for rendering triangles rather than voxels. Therefore the usual approach is to create a triangle mesh which has the same shape as the underlying voxels and then render this instead. This triangle mesh can also be used for other purposes such as collision detection.

Cubiquity largely hides this implementation detail from the user. All you need to do is write the desired values into the voxels, and Cubiquity automatically takes care of keeping the corresponding mesh synchronized with them. These voxel values may be read from disk, be generated procedurally, or created through other approaches. This is largely up to you.

Understanding Cubiquity

In general you should not need to know much about the underlying Cubiquity engine as most important details are hidden behind the C# scripts, but this section discusses a couple of details which are worth keeping in mind.

Calling the Cubiquity Native-Code Library

Functions defined in the native-code library can be called from Unity3D scripts using some magic known as P/Invoke. The file 'CubiquityDLL.cs' uses this P/Invoke technology to provide thin .NET wrappers around each function which is available in the Cubiquity engine. The rest of the Cubiquity for Unity3D scripts are then implemented in terms of these wrapper functions. As a user of Cubiquity for Unity3D you are unlikely to come across these implementation details unless you deliberately go exploring the code (which of course you are welcome to do).

Using native-code allows for some significant performance and memory optimizations compared to systems implemented using just Unity3D scripts. However, it also imposes some limitations, in particular that it cannot be used with the Unity3D web-player because this does not support native code. The native-code library also has to be compiled separately for each platform on which it needs to run (Windows, OSX, Linux, etc), but this is an issue which we as developers have to deal with rather than you as a user.

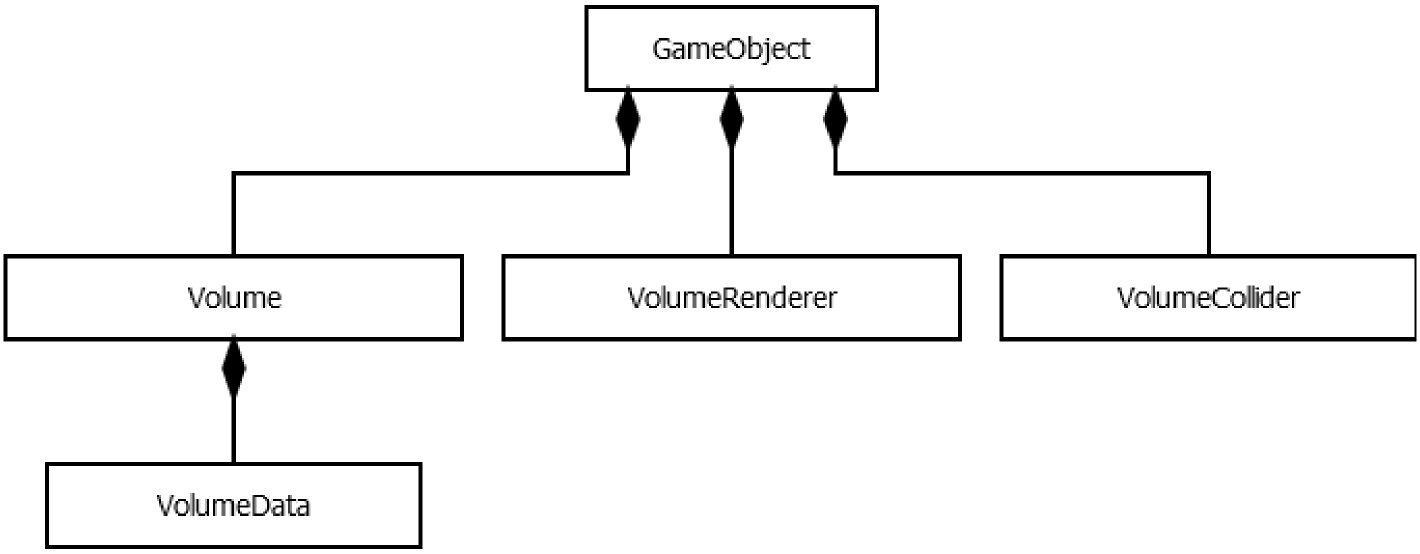
The Cubiquity Voxel Database Format

Voxel environments can get very large (potentially containing billions of voxels) so efficient storage of such worlds is of utmost importance. The Cubiquity voxel engine stores a volume as a *Voxel Database*, which is a single file containing all the voxels in the volume. Internally it is actually an SQLite database and so can be opened with a tool such as SQLite Browser. Such as tool will let you view certain properties of the volume such as its dimensions, but you won't be able to gain any meaningful insight into the voxel data itself as it is stored in an efficient compressed format.

Understanding Cubiquity for Unity3D

As mentioned previously, *Cubiquity for Unity3D* is the integration layer which connects our Cubiquity voxel engine to Unity3D. It does this by wrapping voxel databases as Unity3D assets, converting the generated mesh to the the required format, and providing Unity3D materials to control the appearance of your voxel objects. These and other aspects of the system are discussed in this section.

We have tried to make the design and interface as consistent as possible with the approaches used elsewhere in Unity. To this end we have adopted a compoent-based model in which the user can add a GameObject to a scene, give it a 'Volume' component to make it into a voxel object, and then add VolumeRenderer and VolumeCollider components to control it's behaviour. With this in mind, the structure of a typical volume is as follows:



The structure of a typical volume

Note that the components shown are actually the base classes and in practice are never used directly. Instead you will use a particular set of subclasses (such as TerrainVolume, TerrainVolumeData, TerrainVolumeRenderer and TerrainVolumeCollider) depending on the kind of volume you would like to create. We now look at each of these components in more detail.

The Volume Component

Adding a Volume component to a GameObject makes it into a voxel object. The Volume class does not provide much functionality on its own as most behaviour (rendering, physics, etc) is implemented in the other related components. Instead the class acts more to tie the other components together.

Volume components also have custom inspector implemented which allow you to edit the volume in an intuitive way. For example, the TerrainVolume custom inspector exposes the sculpting and painting tools which can be used to build your own terrain in the Unity3D editor. The inspector also has a 'Settings' panel where you can choose which volume data is being used.

The VolumeRenderer Component

The visual appearance of the volume is controlled by the VolumeRenderer, and if a VolumeRenderer is not present then the volume will not be visible in the scene. The role is similar to that of the MeshRenderer for Unity's Mesh class, and this is no coincidence considering that meshes are used internally by Cubiquity to display the volume data. The VolumeRenderer exposes properties including the material of the volume and its shadowing behaviour.

The VolumeCollider Component

The VolumeCollider component should be attached to a volume if you wish a collision mesh to be generated. This will allow other object in the scene (such as rigid bodies) to collide with the volume. There are currently no properties (the VolumeCollider is simply present or not) but these will likely be added in the future.

The VolumeData Component and Cubiquity Assets

Cubiquity for Unity3D wraps voxel databases with a class called VolumeData, and more specifically with its subclasses called TerrainVolumeData and ColoredCubesVolumeData. These are very thin wrappers, which basically just store the filename of the voxel database and provide functions to get and set the voxel values.

Because VolumeData is derived from Unity's ScriptableObject class it is possible to turn it into a Unity Asset. You can then select the asset in the project view to see the path to its voxel database in the inspector, and you can drop it onto the 'Volume Data' field which is found under 'Settings' in the Volume inspector.