ZARENIKIT

Voxel Framework

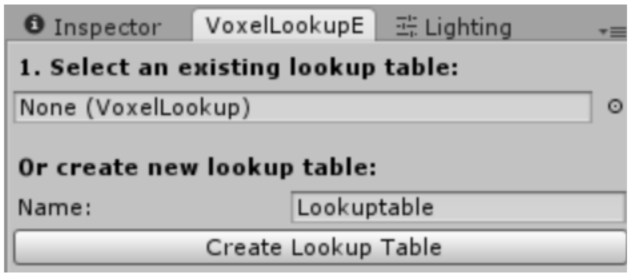
User Manual

Getting Started

Starting a project using ZareniKit Voxel Framework is fairly easy. This guide will walk you through the basics of setting up a "Chunk," the grid on which your voxels exist, and populating it with voxels.

{PART 1} BUILDING A LOOKUP TABLE-

In the ZareniKit Voxel Framework, all voxels are of a specific type. These types are defined in a Lookup Table, which stores all of the data associated with a voxel type, such as its solidity, or the texture it has. To start using voxels in your project, you will first need to create a lookup table. ZareniKit makes this easy by providing a lookup table editor. To open this editor, go to *Tools>ZareniKit>Voxel Lookup Editor* in the toolbar. You should see an editor window pop up. This is the lookup table editor. The window is designed to be docked vertically like the inspector, although it is entirely up to you to fit it with your layout. This window is where you will create all of your voxel types. You can only have a maximum of 65,535 types, although there are very few situations in which one would need more types than that. To start off, create a new lookup table by first typing in the name of the table into the box labelled 'Name,' and then pressing *Create Lookup Table*. You have just created a lookup table. You will notice that the object field also selected your table. To open your table in the future, simply drag the asset into the box at the top of the window, or click the circle on the side to select your table from a list of all created tables. Once your table is selected, simply click *Open Table*, then *Start Editing*. You should see the properties of the first voxel in your table. The first voxel has an ID of 0, and is the default voxel that is used if a voxel would otherwise be null or out of range, so it is recommended that you use this as an 'air' or 'empty' voxel. First, give it a good name, like 'Air.' Air does not need to be visible and is definitely not solid, but it is transparent, so check that box. Air is not visible, so it doesn't need a texture. You can ignore the Voxel Type and Instance Attributes for now. You have now just made your first voxel type. A world made solely out of air would not be much to look at, though. We are going


Inspector | VoxelLookupE | Lighting

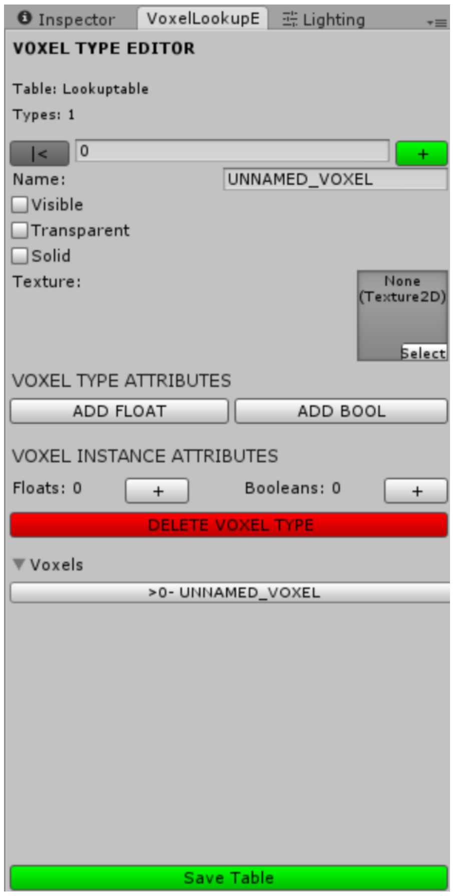
1. Select an existing lookup table:

None (VoxelLookup)

Or create new lookup table:

Name: Lookuptable

Create Lookup Table


Inspector | VoxelLookupE | Lighting

VOXEL TYPE EDITOR

Table: Lookuptable

Types: 1

|< 0 +

Name: UNNAMED_VOXEL

☐ Visible

☐ Transparent

☐ Solid

Texture: None (Texture2D) Select

VOXEL TYPE ATTRIBUTES

ADD FLOAT | ADD BOOL

VOXEL INSTANCE ATTRIBUTES

Floats: 0 + Booleans: 0 +

DELETE VOXEL TYPE

▼ Voxels

>0- UNNAMED_VOXEL

Save Table

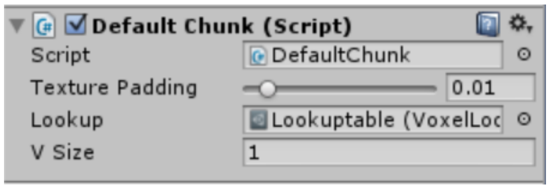
to need another voxel type. Click the green + to the right of the voxel id. This creates a new voxel type. You can cycle through all of your voxel types by pressing the left and right arrows on the sides of the voxel id, enter the id directly, or select a voxel by name from the foldable list of voxels below the delete button. Let's create a ground block. The ground should be visible and solid but not transparent. The ground also needs a texture. Pick any texture you want the ground to look like and drag it into the texture slot. Note that the texture must be read/write enabled. To set a texture to be read/write enabled, set it to type *Advanced* in its import settings, then check the box labelled 'Read/Write Enabled' before hitting *Apply*. The texture will now work properly with your voxel types. Now that you have these two voxel types, you can go ahead and press Save Table at the bottom of the window. This will save your table as well as automatically create a texture atlas from your voxel textures. Your voxel lookup table has been successfully created.

{PART 2} SETTING UP MATERIALS-

Now that you have a lookup table, you will need to make materials for your chunk to use when displaying the voxels. Chunks in the ZareniKit Voxel Framework use two materials- one for opaque voxels, and one for transparent voxels. Currently our lookup table doesn't have any visible transparent voxels, so we don't necessarily need the transparent material, but we will make one to make things easier later on. The names of these materials do not matter, but the materials you create should only be used on chunks using the same lookup table. Go ahead and create two materials and a GameObject to be your chunk. The opaque material should be assigned to Element 0 of the GameObject's mesh renderer, and the transparent one to Element 1. You don't need to give these materials a texture, as the chunk will take care of that for you.

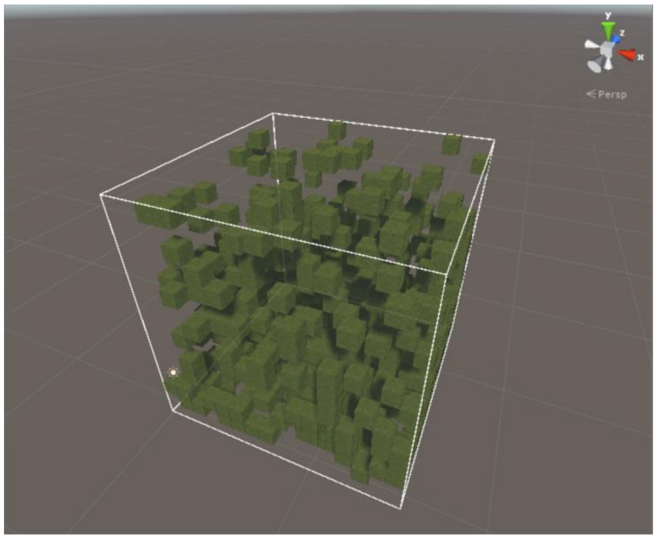
{PART 3} CREATING A CHUNK-

Creating a chunk is very simple. To create a basic chunk, simply select a GameObject and add the 'Default Chunk' script to it. This script can be found in *Components>Voxels>Default Chunk*, and is the simplest chunk possible, assigning random voxel types to each point. Upon adding the chunk script, you should see a small list of properties, and should notice that a mesh filter, renderer, and collider have been added if they were not already there. The first property is the Texture Padding slider. This property is available on every chunk, and will be automatically included in any custom chunks you make. The texture padding slider controls how much of each voxel's texture should be disregarded. If this value is too high, then the textures will appear to be stretched or enlarged. If this value is too low, then you may begin to see black lines appear between voxel faces. At its default value of 0.01, most textures will continue to appear seamless without black


Default Chunk (Script)

Script	DefaultChunk
Texture Padding	0.01
Lookup	Lookuptable (VoxelLoc
V Size	1

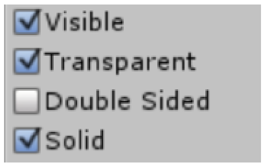
lines appearing. If this is not the case for your voxels, then change the slider until you achieve the desired effect. The next property is the lookup table. Drag your lookup table from part 1



into this slot. This way, your chunk knows what voxel types to use. The final parameter is the voxel size. The value you use here depends entirely on your game and what you are using the voxels for. A setting of 2.5 is good for walking around in comfortably with the default FPS controller. Now, it's time to press play. Your chunk should now be filled randomly with the voxel types you defined in your lookup table. If you drag in an FPS controller, it is possible to walk on the chunk. If you hit pause, you should see something similar to the image on the left.

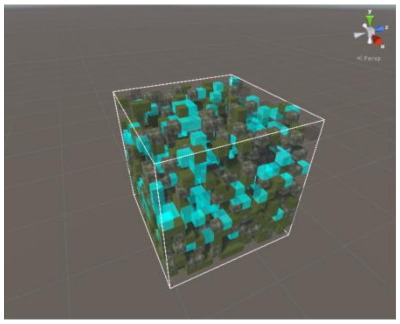
{PART 4} NEW VOXEL TYPES and THE CHUNK MESH-

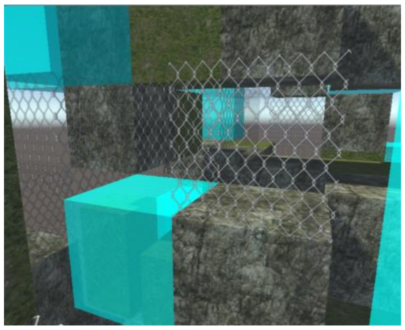
Currently the chunk looks rather boring. Fortunately, the ZareniKit Voxel Framework lookup table editor can solve that problem. Want to add rocks or sand to the environment? How about liquids or gasses? Let's go back to the lookup table editor you used in part 1. Select your lookup table and open it up. Adding rocks or sand is easy. Simply hit the green + button to



- ✓ Visible
- ✓ Transparent
- ☐ Double Sided
- ✓ Solid

create a new voxel type and name it sand. Check the Visible and Solid boxes, and give it a read/write enabled texture. This is easily done and can be repeated as desired for any other voxel that behaves in a similar way. Glass can also be easily created, by making a voxel that is visible and solid, but also transparent, and then applying a transparent texture. This will use the second material you made, so you can use a refractive shader as well if desired. Also, you will notice a checkbox appear to allow you to make the voxel Double Sided. This means that your voxel will be visible from the inside as well. Generally it is a good Idea to leave this false, as double-sided voxels make the chunk mesh more complex. Liquids and gasses can be created in a similar way, while leaving the Solid checkbox off. Now that your lookup table contains more voxel types, your chunk should look a bit more interesting. This chunk now has multiple types of ground voxels, as well as transparent, non-solid blue voxels and even fence voxels that appear transparent.





Making Your Own Chunks

ZareniKit Voxel Framework allows you to extend its chunks through scripting for full control of your voxels' behavior and generation.

{PART 1} INTRODUCING THE ZKAPI-

Extending and using the ZareniKit Voxel Framework can be done through scripting can be done through ZareniKit's API, the ZKAPI. To use the ZKAPI, simply add `using ZKAPI` to your scripts. The ZKAPI contains all the classes and functions that you will need to create your own custom chunks, automators, and functionality for your voxel worlds. Read the API documentation included with this package for in-depth details about the ZKAPI.

{PART 2} CREATING A NEW CHUNK WITH SCRIPTS-

Custom chunks are simply scripts that inherit from ZKAPI.ChunkObject. To create a custom chunk, create a new script in your preferred scripting language, and import the ZKAPI namespace as well as inherit from ChunkObject, rather than MonoBehaviour. Congratulations, you have just created a custom chunk. Drag this onto your chunk GameObject and remove any other chunk scripts. You should notice that this chunk does absolutely nothing. This is because the chunk's grid has not been set up yet, and no voxels have been generated. There are a few methods that you will need to invoke. The first is `Initialize()`. This is often best put in the `Awake()` event of your custom chunk script. `Initialize()` takes anywhere between 1 and 5 parameters. The first parameter is required, and is the lookup table you will be using. Create a public variable of type LookupTable , and use that value for the lookup table argument. The next three arguments are the size of the chunk in voxels. By default, these are set to 16x16x16 voxels, which is a good size for most chunks. The final argument is the voxel size. This method works similarly to scaling, but is taken into account by the mesher rather than scaling the entire object. This is nice to use if you are making a large grid of voxels. The chunk and voxel sizes are optional parameters.

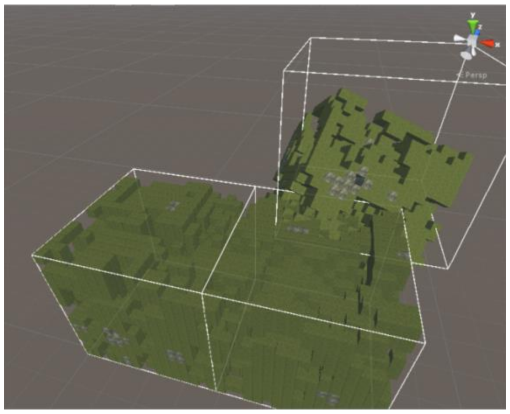
If you run the game now, the chunk will still be empty, because no voxels have been generated. This is easily solved by calling `Generate()` any time after the chunk has been initialized. You will have to specify an IVoxelGenerator. You can make your own, or you can use a new VoxelRandomGenerator instance. If you do this, and have specified a lookup table, your chunk should behave just like the DefaultChunk script. Of course, there is a lot more you can do here, including making your voxel move around or interact with each other.

{PART 3} USING VOXEL GENERATORS-

Chunks are populated with voxels by using voxel generators. Voxel generators are classes that implement the interface `IVoxelGenerator`. `IVoxelGenerator` has a single method, `Sample()`, that is used to determine the voxel at a given point. Currently, the custom chunk you made uses a generator called VoxelRandomGenerator, which assigns a random voxel to each point, where lower y values are less random. This generator is merely a placeholder and is not very useful for an actual game. If you want, for instance, perlin noise, or to sample a 3D texture, you will need to create a new voxel generator. To do this, simply create a new script with a class that implements `IVoxelGenerator` and add a method named Sample that returns an int and takes a Vector3 as an argument. The Vector3 is the position of the voxel that must be assigned, and the int is the id of the voxel. Here is an example of a more interesting generator:

```
public class MoreInterestingGenerator : IVoxelGenerator{
      public int Sample(Vector3 pos){
            return Mathf.RoundToInt(Mathf.Clamp01(
            Mathf.PerlinNoise(pos.x/8f,pos.y/8f)
            *Mathf.PerlinNoise(pos.y/8f,pos.z/8f)
            *Mathf.PerlinNoise(pos.z/8f,pos.x/8f))
            *7f);
      }
}
```

This generator uses Unity's built-in perlin noise functions to create a much more interesting chunk shape. The input of your sample function is in world coordinates, so chunks placed side-by-side will appear continuous, and the shapes of chunks that are rotated or scaled will also appear continuous, even with differently rotated or scaled chunks. The generator you create will depend on your use for the chunk.



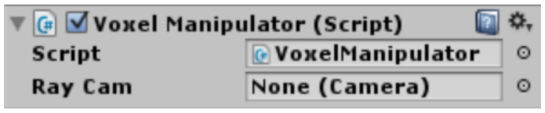
Making Interesting Voxels

The ZareniKit Voxel Framework allows chunks to be changed in code, so your voxels can take on a number of interesting behaviors.

{PART 1} CHANGING CHUNKS IN CODE-

There are a few ways to change a chunk at runtime using code. The first is to make a custom chunk. In the custom chunk, you are free to change the chunk in `Update()`, `Awake()`, and any of the other functions. It should be noted, however, that if code is put in `LateUpdate()`, you must call `base.LateUpdate()`, or the chunk may not update its mesh reliably. Use `GetVoxel()` and `SetVoxel()` to modify the voxels in your chunk.

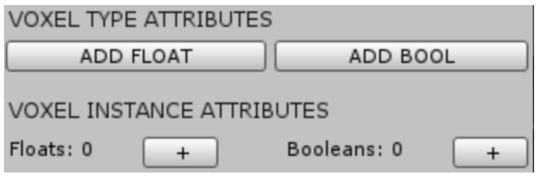
The second is to make a new VoxelBehaviour. This is a script that inherits from VoxelBehaviour, and is used to change the voxel in a chunk in a similar manner to how Unity's own MonoBehaviour can change an object. There are a number of messages that a VoxelBehaviour can receive, including when a collision is detected or the mouse buttons are clicked. Most of these messages can pass a VoxelData parameter, which gives information about the voxel that is currently being clicked, collided with, etc. For many of the VoxelBehaviour's functions to work, you must have a Voxel Manipulator enabled in the scene. The Voxel Manipulator can be found under *Component>Voxels>Voxel Manipulator*. The Voxel Manipulator has a single property, Ray Cam, that is a reference to the camera that the manipulator will be using to determine where the player has clicked. If this is set to a camera in the scene, then rays will be cast from the center of the viewport of that camera. If it is not, then the script will attempt to use a camera attached to the same GameObject that it is on. If the GameObject it is on does not have a camera attached, if use the currently active camera. If there are no cameras in the scene, or none are active, the manipulator will disable itself. This script uses the mouse buttons to change voxels. by clicking them:


Voxel Manipulator (Script)
Script: VoxelManipulator
Ray Cam: None (Camera)

```
public class EditableVoxels : VoxelBehaviour {
	void OnLMBDown(VoxelData v){
		chunk.SetVoxel(0,v.coords); //Set the voxel to type 0 (air) if 
left-clicked.
	}
	void OnRMBDown(VoxelData v){
		chunk.SetVoxel(3,v.backCoords); //Set the voxel to type 3 
(stuff) if right-clicked.
	}
	//Chunk meshes will automatically update after a call to SetVoxel.
	//You can also perform operations with other mouse buttons, using 
functions such as:
	//OnLMBUp, OnRMBUp, OnMMBDown, and OnMMBUp, as well as with collisions 
using
	//OnVoxelCollisionEnter and Exit.
}
```

{PART 2} USING VOXEL TYPE AND INSTANCE ATTRIBUTES-

Sometimes, you might want a particular voxel or set of voxels to behave differently. You could check what voxel you are dealing with directly using if statements in your custom chunk or VoxelBehaviour, but if you have many types of voxels that share a common behavior, this can get tedious and messy. Fortunately, the ZareniKit Voxel Framework has a solution: Voxel Attributes. Voxel attributes are values that are given to a specific voxel or voxel type. To edit voxel attributes, open up your lookup table in the lookup table editor. Between the texture property and the delete button, you should find a section devoted to editing voxel type attributes.



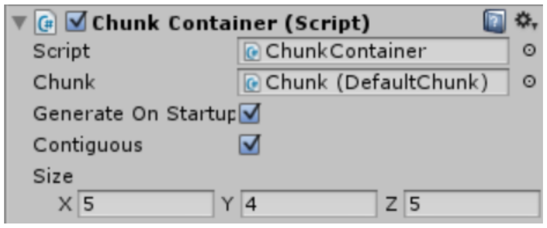
There are two types of voxel attributes: voxel type attributes and voxel instance attributes. Voxel type attributes are the simplest kind. These values are stored on the lookup table, and behave similarly to the Visible and Transparent properties that you are already familiar with. When you add a voxel type attribute, every voxel type can have its own value for that attribute. Clicking Add Float or Add Bool will create a voxel type attribute. You can then give it a name, and each voxel type can give it a value. The attribute can be accessed in code either by its name or its ID number using `GetVoxelTypeBool()` and `GetVoxelTypeFloat()`. Voxel type attributes exist only in the voxel lookup table, so adding them should not have any noticeable impact on memory usage. Voxel instance attributes, on the other hand, are stored individually in each voxel. These are useful for things that differ on a per-voxel basis, such as damage or time until an action takes place. The number of voxel instance attributes can be changed for each voxel type in the lookup table editor. Voxel instance attributes exist for each voxel, so adding them can cause a significant difference in memory consumption, particularly in large grids and with many float attributes. To conserve memory, voxel instance attributes have no name, and can only be accessed in code by their ID number using `GetVoxelInstanceBool()` and `GetVoxelInstanceFloat()`. Unlike voxel type attributes, however, voxel instance attributes can be set in using `SetVoxelInstanceBool()` and `SetVoxelInstanceFloat()`.

{PART 3} MAKING VOXELS INTERACT WITH THE WORLD-

Chunks in the ZareniKit Voxel Framework can interact with other objects in your scene just as they would normally. Chunk collisions are typically done with a mesh collider, and can interact with other physics-enabled objects this way. For most other objects in your scene, the chunk is a completely ordinary GameObject. Individual voxels, on the other hand, do not behave like ordinary GameObjects, and in fact are not GameObjects at all. Individual voxels, unlike chunks, cannot move independently of the chunk. It is possible, however, to simulate voxels interacting with the world using VoxelBehaviours. VoxelBehaviours allow you to modify chunks during the `OnVoxelCollisionEnter()` and `OnVoxelCollisionExit()` messages. These are similar to the standard Unity Monobehaviour's `OnCollisionEnter()` and `OnCollisionExit()` messages, but rather than give information about the collision itself, it gives information about the voxels that were hit through a VoxelData object, similar to when a voxel is clicked. Chunks and VoxelBehaviours also allow you to change between chunk and world coordinate systems by using `ChunkToWorldCoords()` and `WorldToChunkCoords()`. These utilities are useful for debugging, simulation, generation, and the behavior of voxels.

{PART 4} USING CHUNK CONTAINERS-

Sometimes you might want several chunks in your scene. If you are making something like a terrain, you might want hundreds of chunks in your scene. You could create and position hundreds of chunk prefabs by hand and add individual scripts to control them, and fiddle with your player so that they can interact with the right voxel in the right chunk, but there is a much easier way of creating large numbers of chunks, and that is the Chunk Container. The Chunk Container can be found under *Component>Voxels>Chunk Container*, and has a few simple properties. The first is the Chunk property, where you drag in a prefab of the chunk object that you want to use. The second is the Generate on Startup property. If checked, the Chunk Container will create its own chunks when the game starts. The Contiguous property determines how the chunks will access other chunks' voxel data. If all of your chunks will be generated side-by-side and remain that way, checking this property will result in faster voxel lookups. If your voxels are not generated contiguously, or will move from their position at any point in the game, it is best to leave this box unchecked. This results in slower voxel lookups, but more consistent behavior. It is recommended that non-contiguous chunk containers contain only a few chunks. The final parameter is the number of chunks that the container will hold in each direction. These values are rounded to the nearest integer.


Chunk Container (Script)
Script: ChunkContainer
Chunk: Chunk (DefaultChunk)
Generate On Startup: ✓
Contiguous: ✓
Size
X 5 Y 4 Z 5

When the Generate on Startup property is checked, your chunks will attempt to use their generators after all chunks have been destroyed, therefore if you are using a custom chunk that generates at a separate time, you may end up generating the voxels in your chunk multiple times.