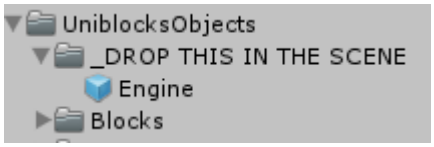
UNIBLOCKS USER MANUAL
Version 1.4.1

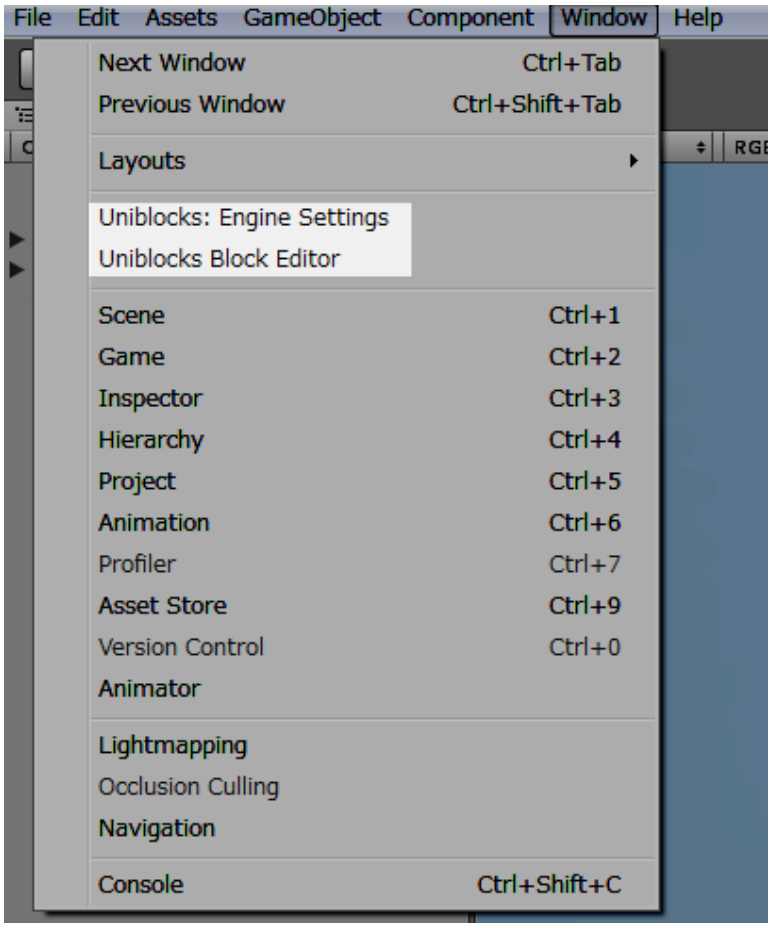
Index

1. Setting up the engine

To get started, drop the Engine prefab into your scene. You can now configure the engine using the editor windows:



Use the editor windows to configure the engine:
Window -> Uniblocks Engine Settings to configure the basic engine settings such as chunk size and view distance.
Window -> Uniblocks Block Editor to add or modify blocks.



You can keep the default settings for now, and tweak them later as needed.
In addition to the Engine prefab, you'll also need something to initiate the spawning of terrain chunks in the scene. The package comes with a Player prefab which has a ChunkLoader component attached. ChunkLoader simply tells the engine to load chunks around the position of the object it's attached to (in this case, the Player). You can drop this prefab into your scene, or else attach the ChunkLoader script to your own object. Alternatively, you can call the static function Uniblocks.ChunkManager.SpawnChunks(Vector3 position) from anywhere in your scripts to load the chunks at the given world position.

Once the Engine prefab is properly set up in the scene and ChunkManager.SpawnChunks is called (from ChunkLoader or otherwise), terrain chunks will begin to spawn around the spawn point within a radius determined in ChunkManager.SpawnDistance (the radius is measured in chunks, that is, a radius of 7 means that it will spawn up to 7 chunks in each direction around the origin point).

Note: ChunkManager.SpawnChunks can either take an absolute world position (as three floats (float x, float y, float z), or a Vector3), or a chunk index of the origin chunk (as three ints (int x, int y, int z), or an Index). If you pass it an absolute world position, it will be converted to a chunk index. The chunk's index is directly related to it's position in the world (for example, assuming the chunk's side length is 24, a chunk with the index of (1,0,1) will be situated at the world position of (24.0, 0.0, 24.0).

Once spawned, each chunk takes care of itself, populating it's data array and generating it's mesh.

- if the chunk data exists on the hard drive, ChunkDataFiles will handle it's loading;
- else a terrain generator specified in the Chunk's GenerateVoxelData function (ExampleTerrainGenerator by default) will generate the terrain based on the global variable Uniblocks.Engine.WorldSeed
- when all chunks finish spawning, ChunkMeshCreator will create the chunk's mesh based on the chunk's voxel data.

When SpawnChunks is called at a different location, any currently spawned chunks that are out of range of the new origin point will be destroyed, and their data will be stored on disk.

You can also save all currently spawned chunks using the Uniblocks.Engine.SaveWorld() static function. If you have an object with the Debugger script that comes with this package attached in the scene, you can use the keyboard shortcut 'v' to call this function.

Note: Uniblocks uses the collision layer 26 for blocks which are set to have no collision. On startup the engine will automatically set the layer to ignore collisions with all layers. This can affect other GameObjects in your project which also using layer 26.

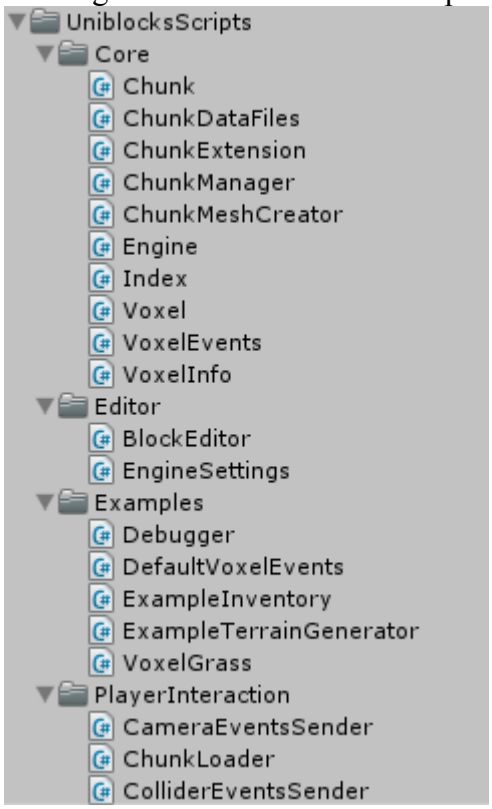
Save file location
By default, the world data is stored in **/root/Worlds/WorldName/**, where **root** is the root directory of the application (or the Unity project), and **WorldName** is the world name (TestWorld by default).

You will need to change the path if you're building your game for mobile use. Consult platform-specific documentation to determine the appropriate folder for storing data.

In order to change the save file location, open the Engine.cs script, then find and edit the line commented with “// you can set World Path here”.

Engine Structure

The engine is divided into several parts, corresponding to the folders in the package:



Core: The core of the engine, you generally won't need to touch these, unless you want to modify some core features of the engine.

PlayerInteraction: These scripts send events such as OnMouseDown to the core VoxelEvents class. This is separate from the core of the engine because the model of player interaction with the terrain might vary depending on what kind of game you're making, so you might want to modify these events or perhaps even write your own. However, the default PlayerInteraction scripts should cover some of the most common situations, so perhaps you won't need to touch these at all.

Examples: These are example implementations of various gameplay mechanics. You'll probably not be using them unmodified in your actual game. They're included for demonstration purposes.

Editor: Scripts implementing the Engine Settings and Block Editor windows.

Terrain Generation

Terrain generation is done by a TerrainGenerator attached to the Chunk prefab. By default the engine uses the ExampleTerrainGenerator script. To customize the generation of the terrain, you can either this script, or create your own script inheritting from TerrainGenerator, and attach it to the Chunk prefab.

This manual will not go into specific implementations of terrain generation as this is a wide topic in and of itself, though you can check the ExampleTerrainGenerator for a very basic example to get you started.

Some things to keep in mind when writing a custom terrain generator:

- Your script should inherit from TerrainGenerator, and the terrain generating function should override the GenerateVoxelData function.

- Each chunk has it's own instance of terrain generator. You can access the specific Chunk component of the current chunk through the local variable *chunk* (inherited from TerrainGenerator).

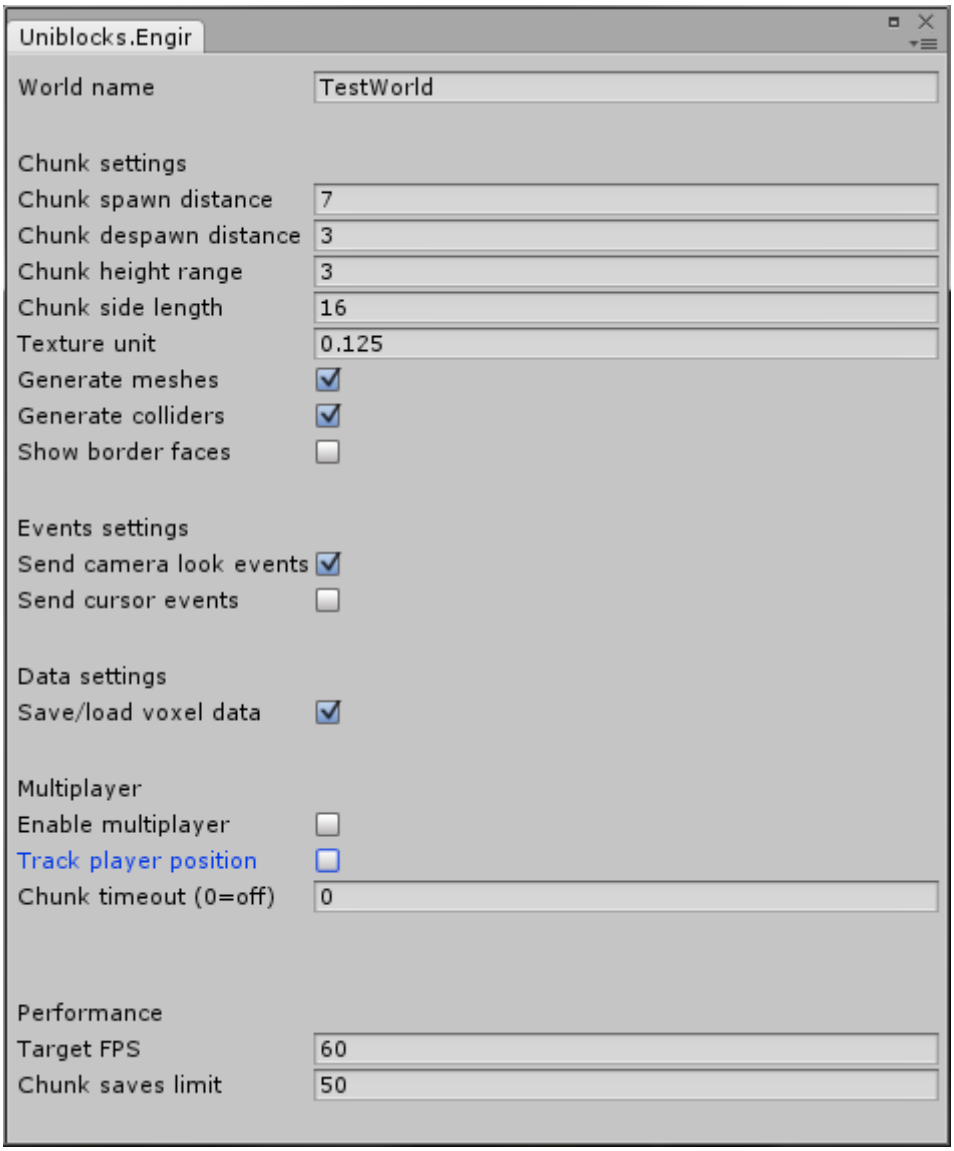
- You can access the random seed through the local variable *seed* (inherited from TerrainGenerator).

- You can get the absolute world position of each voxel within the current chunk using Chunk.VoxelIndexToPosition.

- In your terrain generation script you should be using Chunk.SetVoxelSimple to pass the generated data to the chunk. SetVoxelSimple is slightly faster than SetVoxel since it doesn't need to check whether the voxel is out of the chunk's bounds.

2. Engine Settings

The Engine Settings window controls the global settings such as chunk size or world name. You might want to be changing some of these settings during runtime depending on user input (for example, changing the view distance), others should be set only once and remain fixed at runtime (for example, the size of the chunks). Note that the Engine Settings window cannot be used during runtime, you will need to use custom scripts to change the settings at runtime instead.



You can access the Engine Settings window through the Window menu.

World name: The name of the currently active world. The voxel data will be saved and loaded to and from a folder corresponding to the world name (/root/Worlds/WorldName/ by default). You can change the world name at runtime using the function Uniblocks.Engine.SetWorldName(string worldName). (Default: TestWorld)

Chunk spawn distance: The horizontal distance (in chunks) from the origin point within which chunks will be spawned. A chunk distance of 7 (Default) will spawn 7 chunks in all four horizontal directions around the origin point. You can change this at runtime by changing the Uniblocks.Engine.ChunkSpawnDistance int. (Default: 7)

Chunk despawn distance: Additional distance required to despawn chunks. Chunks will not despawn until their distance from the origin point exceeds the chunk spawn distance plus the additional despawn distance.

Chunk height range: The maximum positive and negative vertical chunk index of spawned chunks. A chunk height range of 3 (Default) will only spawn a chunk as long as the y coordinate of it's index is between 3 and -3. You can change this at runtime by changing the Uniblocks.Engine.HeightRange int. (Default: 3)

Chunk side length: The length of the side of each chunk (in voxels). For example, a side length of 16 (Default) means that all chunks are 16x16x16 cubes. Saved world data is only compatible with the chunk side length it was saved with originally, so changing this setting will result in previously saved data (with different chunk size) to be incompatible. Therefore you should decide on one chunk size, and never change it during runtime. The chunk side length does NOT have to be a power of 2 (for example, 17 is a valid side length). (Default: 16)

Texture unit: The ratio of side length of one block's texture to the side length of the texture sheet. To calculate this, simply divide the texture side length by the texture sheet side length. For example, the default texture sheet size is 512x512, and each texture within it is 64x64, so we divide 64 by 512 to get 0.125. (Default: 0.125)

Texture padding: The size of the padding (unused area) between individual block's textures on the texture sheet. Adding some padding can remove lines appearing between blocks when using certain graphics settings. The texture padding value is interpreted as a fraction of an individual block's texture (for example, a value of 0.1 will ignore 10% of the texture on each side). For best results, it is recommended to use a fraction which matches the size of the pixels (for example, for 1 pixel of padding for a 32 pixel texture, use the value 0.03125 (1 divided by 32)). (Default: 0)

Generate colliders: If true, standard Unity mesh colliders will be generated for all the blocks. You can disable this if you don't want to use the built-in physics system. Disabling the colliders will disable all standard collision detection and significantly boost performance, but you won't be able to use the standard events which depend on raycasts (such as OnMouseDown, OnLook, OnBlockEnter, etc). (Default: true)

Show border faces: If true, the side faces of blocks will be visible if a chunk bordering them is not instantiated. (Default: false)

Send camera look events: If true, the CameraEventsSender component will send events (OnLook, OnMouseDown, OnMouseHold, OnMouseUp) to the block in the center of the field of view of the main camera. Use these if you're using a first-person targetting camera setup. (Default: true)

Send cursor events: If true, the CameraEventsSender component will send events (OnLook, OnMouseDown, OnMouseHold, OnMouseUp) to the block currently under the mouse cursor. Use these if you want the blocks to be clickable with the mouse cursor. (Default: false)

Save/load voxel data: If false, voxel data will not be saved or loaded from hard drive. Chunks will instead always generate new voxel data when they're spawned. Enabling this at runtime will not erase already existing voxel data. It is recommended to disable this setting for a multiplayer client. (Default: true)

Enable multiplayer: Turns the multiplayer functionality on or off. When multiplayer is enabled, client-side chunks will attempt to download voxel data from a server instead of loading or generating it, and any voxel changes made through the Voxel.DestroyBlock, Voxel.PlaceBlock, and Voxel.ChangeBlock will be sent to other connected players.

Track player position: If true:
Server: the server will check for the player's position to determine whether a voxel change needs to be sent to that player.
Client: the client will send a player position update to the server through the ChunkLoader script. (Default: true)

Chunk timeout: If a chunk spawned through ChunkManager.SpawnChunk hasn't been accessed for this amount of time (in seconds), it will be despawned (after saving it's voxel data). Requests for voxel data from clients and voxel changes in the chunk will reset the timer. A value of 0 disables this functionality. (Default: 0, recommended for server: 60)

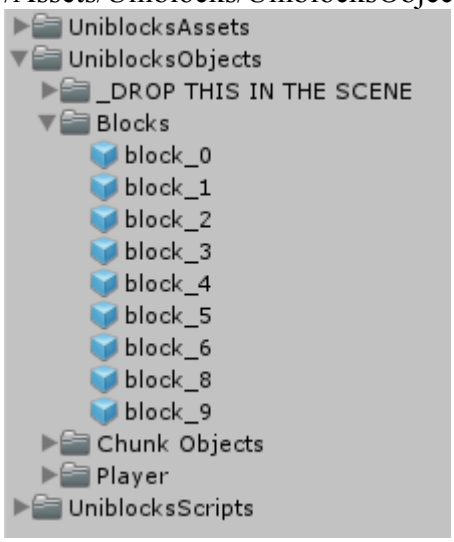
Max chunk data requests: The maximum number of chunk data requests that can be queued in the server for each client at one time (0=unlimited). Reduce this limit if the clients are spawning chunks too fast and you find your server can't keep up with the data requests. (Default: 0)

Target FPS: The engine will balance chunk loading time to maintain the specified framerate. The actual framerate can be slower than the Target FPS setting, depending on external factors, so it might be necessary to set a slightly higher value in order to achieve the desired framerate. (Default: 60)

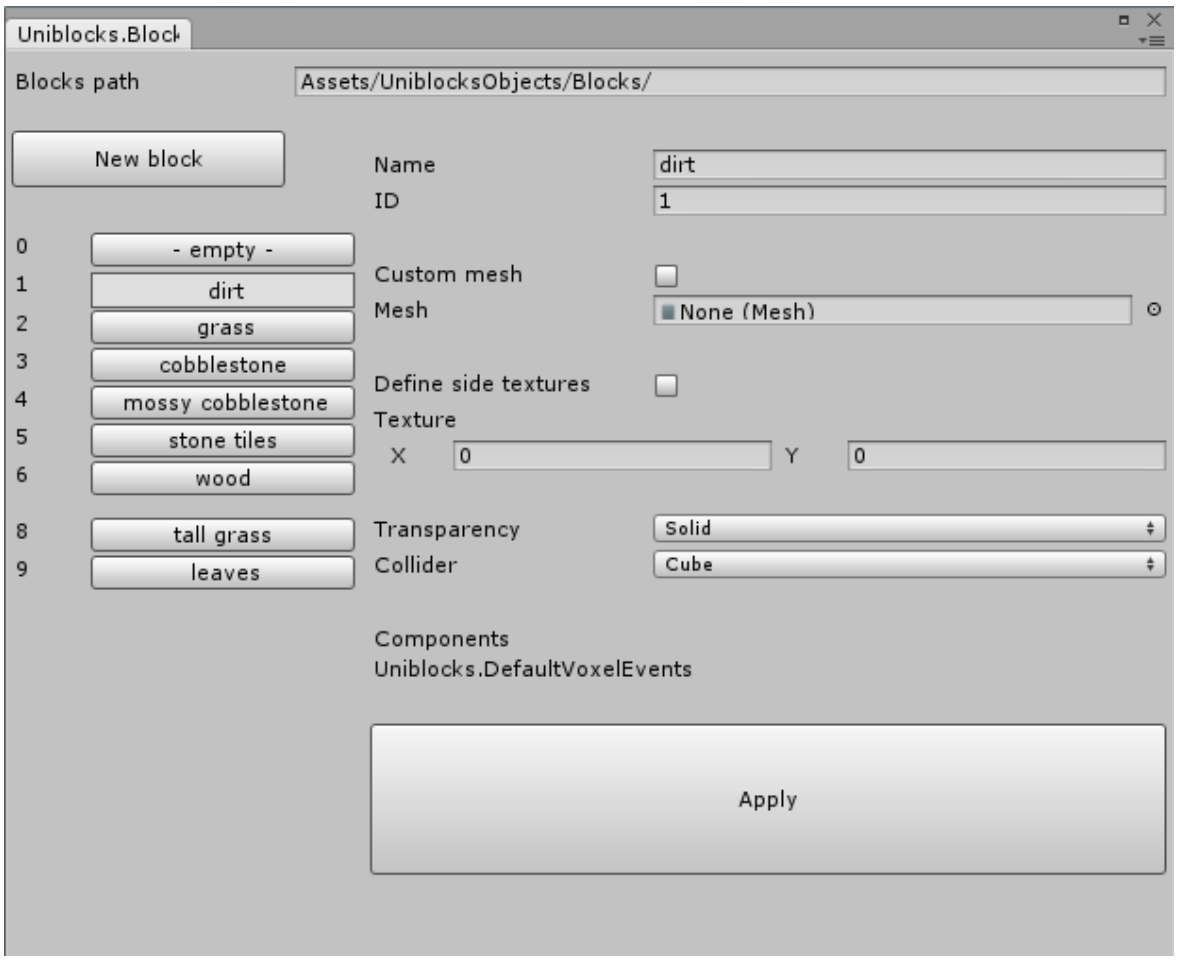
Chunk saves limit: The maximum amount times chunks can save data to disk in a single frame. Chunks save their data when they are being despawned (through the ChunkManager.SpawnChunks function) and when the world is being saved through Engine.SaveWorld. (Default: 50)

3. Block Editor

The Block Editor allows you to add or modify the blocks available in the engine. Each block is a GameObject and is stored as a prefab in your project (by default in /Assets/Uniblocks/UniblocksObjects/Blocks/).



The Block Editor gives you easy access to the properties of each block. It also displays a list of custom components attached to each block, but you will need to use the Unity inspector window to actually edit those. The Block Editor will automatically select the block you're currently editing, so you can access it easily in the inspector. The blocks also cannot be deleted from within the Block Editor. You will need to manually delete their prefabs from your project.



Note: You can edit the blocks in the default Unity inspector window as well, but if you add or remove any blocks, you'll need to hit the Apply button in the Block Editor in order to actually pass the up-to-date block list to the Engine.

Note: Block ID 0 is reserved for the empty block and is required for the engine to function correctly. Please do not delete it.

Blocks path: The path to the blocks prefabs in your Unity project. The default path is Assets/Uniblocks/UniblocksObjects/Blocks/. Make sure the path corresponds to the actual location of the blocks prefabs. If you change the path, close and reopen the window to apply the change.

Block list: The numbers to the left correspond to the voxel ID of the block.
Click on a block's name to edit it. (Note: Any unapplied changes made to the previously selected block will be discarded)
Click “New block” to add a new block to the end of the list. You can also change the ID of an already existing block to create a copy of it.

Each property corresponds to a property of the Voxel class, supplied here in the brackets.

Name: The name of the block. Mostly for convenience. Duplicate names are allowed, so be careful about using these for identification purposes in your code. (string Voxel.VName)

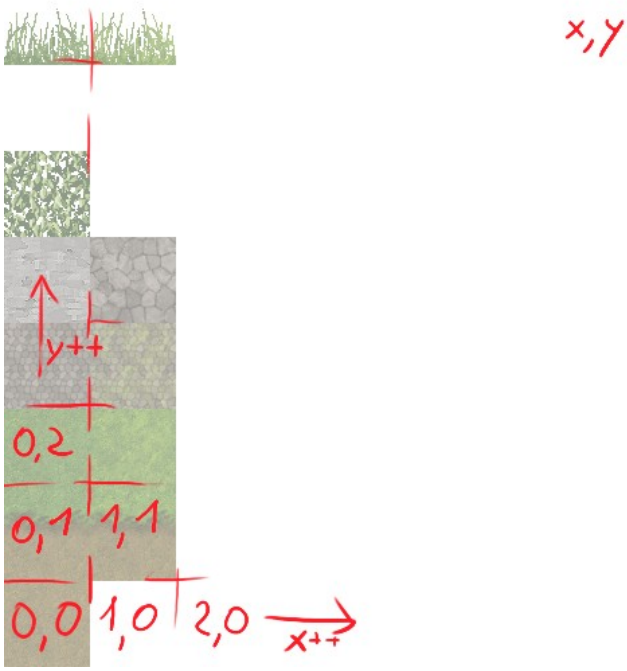
ID: The ID number of the voxel. This is the data that's actually stored in memory to identify a specific voxel. (ushort Voxel.VId)

Custom mesh: Select this if you want to use a custom mesh for this block. This will replace the default cube with the mesh selected in the Mesh section. (bool Voxel.VCustomMesh)

Mesh rotation: Use this setting to apply a rotation to a custom mesh.
none: no rotation
back: 180 degrees
left: 90 degrees to the left
right: 90 degrees to the right
(MeshRotation Voxel.VRotation)

Define side textures: Enable this to specify different textures for every side of the cube. (bool Voxel.VCustomSides)

Texture: The coordinates of the block texture within the texture sheet. The unit of measure is blocks, not pixels, and starts count from bottom left, so for example X=0, Y=3 defines the third block from the bottom in the first column from the left. (Vector2[] Voxel.VTexture)



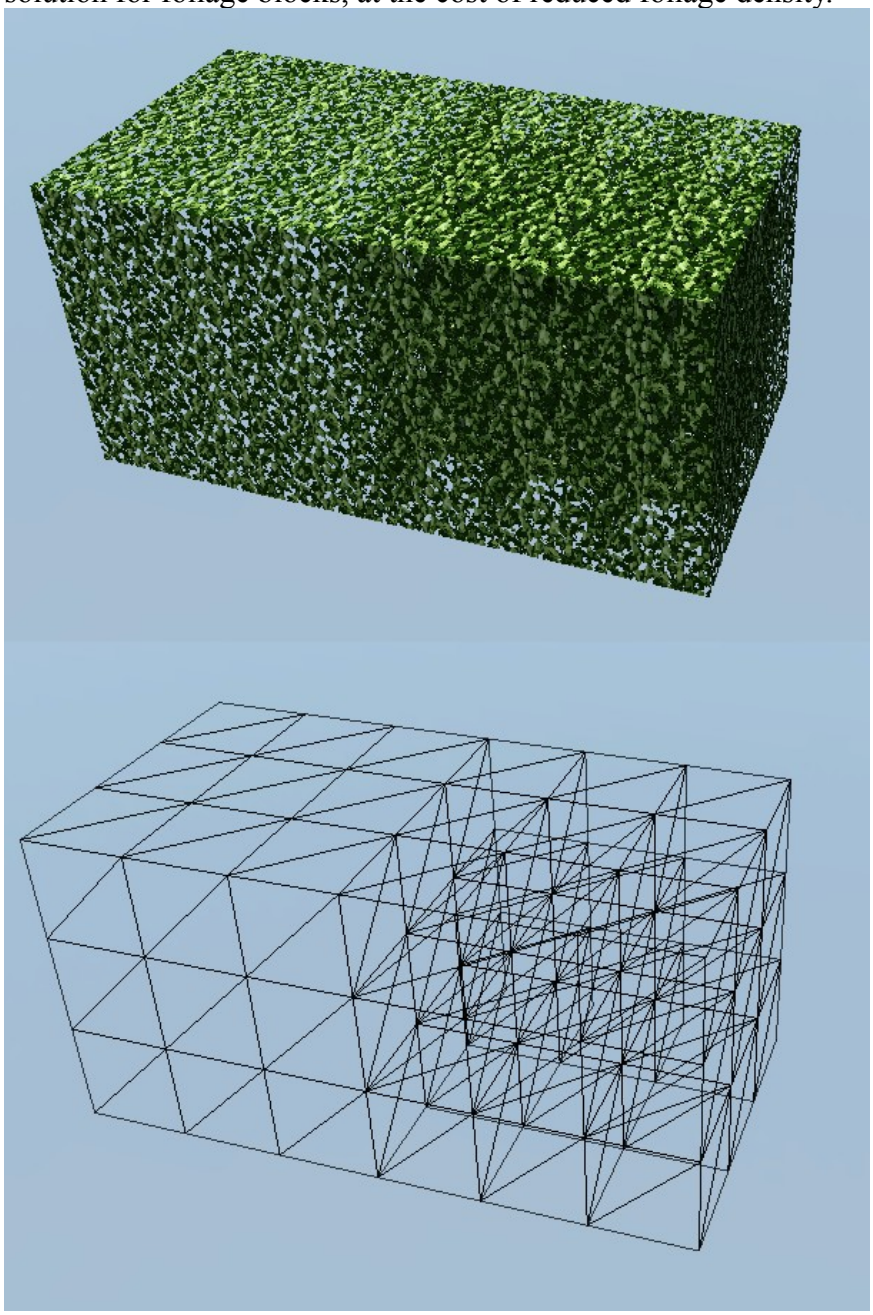
In order to edit the actual textures, you can either modify the texture sheet that comes with the package, or create your own texture sheet and attach it to the Chunk prefab (either replacing it's existing material's texture, or using a new material).

Material index: Determines which material is used for the block's texture. The material index corresponds to materials attached to the Chunk prefab (the Mesh Renderer's Materials array) - for example, a block with a material index of 1 will use the second material, while the default material index of 0 uses the first material. (int Voxel.VSubmeshIndex)

Note: The material index cannot be greater than the number of materials in the Chunk's Materials array.

Transparency: Determines the visibility of the block, depending on the surrounding blocks. (Transparency Voxel.VTransparency)
Solid blocks will obscure all other blocks, meaning that anything surrounded by solid blocks will not be rendered (and any cube side facing a solid block will not be rendered). Use this for standard, non-transparent blocks.

Semi-transparent blocks do not obscure any blocks. An example use might be foliage blocks like leaves, and also most custom meshes. This is the most resource-intensive setting, since it wil always render the entire mesh as long as it's not fully surrounded by solid blocks.
Transparent blocks will obscure only other transparent blocks. A transparent block fully surrounded by other transparent blocks will not be rendered, making this more efficient than the Semi-transparent setting. You might use this for things like glass, or as an alternative, more efficient solution for foliage blocks, at the cost of reduced foliage density.



(left - Transparent; right - Semi-transparent)
Collider: The type of collision for this block. (ColliderType Voxel.VColliderType)
Cube is the default cube collider. A block with a custom mesh can use a cube collider instead of a mesh collider in order to improve performance.
Mesh: use the collider in the shape of the mesh selected in the Mesh section. This only works for blocks using a custom mesh. For blocks not using a custom mesh, a cube collider will be used if this setting is selected.
None: do not use any collider. A cube collider will still be created for this block, but only as a trigger, for purposes of capturing raycasts and trigger events.

Note: Cube collider refers only to the shape of the collider, not an actual Cube Collider type. The chunk always uses a dynamically generated mesh collider. The only thing which determines performance differences between using a cube collider and a mesh collider is the vertex count.

Components: Displays the list of all custom components attached to this block. In order to add or edit the components, use the standard Unity Inspector window.

4. Interacting with the blocks & scripting

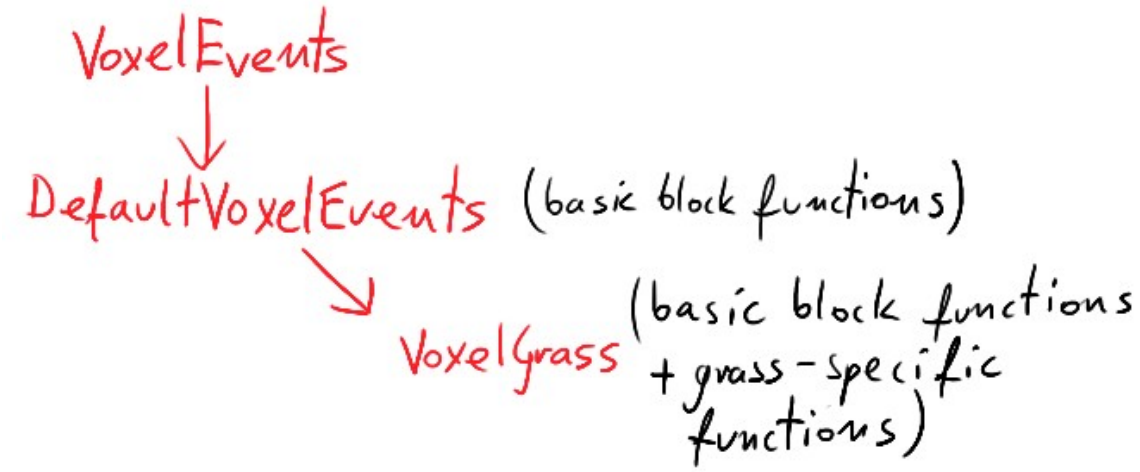
When working with the engine, you will probably want to access the properties of specific voxels. Due to the sheer amount of voxels, they cannot exist in the scene and accessed as individual GameObjects. Instead, accessing individual voxels is done through the VoxelInfo class.

VoxelInfo
Voxels are stored locally in chunks, each chunk containing a finite array of voxel data. Therefore, in order to access a specific, unique voxel, we need to know both it's parent chunk and it's local coordinates within that chunk. VoxelInfo is a data structure which contains reference to a chunk (GameObject VoxelInfo.chunk) as well as the index of a voxel (Index VoxelInfo.index). With this data structure you can store everything you need in order to access a single, unique voxel.

VoxelEvents
VoxelEvents is a class which contains event functions such as OnMouseDown or OnBlockEnter. These functions are called from various other scripts (CameraEventsSender, ColliderEventsSender, and Voxel) on a specific block whenever a corresponding action is captured. For example, when a block is looked at by the player, and the Engine Setting “Send camera look events” is enabled, CameraEventsSender will attempt to call the OnLook function on that block. The block is stored and accessed through the VoxelInfo class, which is an argument of all the event functions.

The VoxelEvents class itself does not contain any code in any of the event functions. In order to implement custom behavior, create a new class which inherits from the VoxelEvents class, override the appropriate functions and implement your custom code there. Then simply attach the new script to a block prefab as a component. Using this system, each block can have a unique component with it's own properties and methods, similar to regular GameObjects.

In addition to custom block-specific code, you might also want to use some more generic functions which will by default apply to all blocks. For example, destroying a block on click might be something common for all blocks. For this purpose, you can use the DefaultVoxelEvents class, provided in the package as an example implementation of the VoxelEvents system. DefaultVoxelEvents inherits from VoxelEvents and implements the default block functions (placing/destroying blocks, turning grass to dirt when a block is placed on top, etc).



An example inheritance structure: All blocks use DefaultVoxelEvents by default, which inherits from the base class VoxelEvents; a few special blocks use custom events on top of the DefaultVoxelEvents, such as the VoxelGrass script, implementing the custom functions as well as the default ones.

Note: Notice how the VoxelGrass class explicitly replicates the "if the block below is grass, change it to dirt" code from DefaultVoxelEvents in the OnBlockPlace function, in addition to it's unique effects. This is necessary because overriding a function essentially ignores all code from the base function, replacing it with the new override code.

DefaultVoxelEvents ← overrides — VoxelGrass

```
public override void OnBlockPlace ( VoxelInfo voxelInfo ) {

    // if the block below is grass, change it to dirt
    Index indexBelow = new Index (voxelInfo.index.x, voxelInfo.index.y-1, voxelInfo.index.z);

    if ( voxelInfo.GetVoxelType ().VTransparency == Transparency.solid
    && voxelInfo.chunk.GetVoxel(indexBelow) == 2) {
        voxelInfo.chunk.SetVoxel(indexBelow, 1, true);
    }
}
```

```
public override void OnBlockPlace ( VoxelInfo voxelInfo ) {

    // switch to dirt if the block above isn't 0
    Index adjacentIndex = voxelInfo.chunk.GetAdjacentIndex (voxelInfo.index, Direction.up);
    if ( voxelInfo.chunk.GetVoxel(adjacentIndex) != 0 ) {
        voxelInfo.chunk.SetVoxel(voxelInfo.index, 1, true);
    }

    // if the block below is grass, change it to dirt
    Index indexBelow = new Index (voxelInfo.index.x, voxelInfo.index.y-1, voxelInfo.index.z);

    if ( voxelInfo.GetVoxelType ().VTransparency == Transparency.solid
    && voxelInfo.chunk.GetVoxel(indexBelow) == 2) {
        voxelInfo.chunk.SetVoxel(indexBelow, 1, true);
    }
}
```

One thing to keep in mind is that you can't store any persistent data in the voxels themselves. The events are called on a temporary copy of the voxel which is destroyed immediately upon completing it's tasks, so any modifications done to the voxel GameObject will not persist. If you need to store persistent data, you can either modify the voxel ID in the main data array of a chunk, or use a custom storage system outside of the voxels.

5. Performance considerations

Note: The performance experienced within the Unity play mode is often significantly worse than in a standalone build. Make sure to take that into account when tweaking performance.

Usually the biggest performance cost comes with loading chunks and building chunk meshes. It can be reasonably assumed that during regular usage the engine will be loading chunks at almost all times, so this is the area where significant performance optimizations can be made.

Loading data
When a new chunk is spawned, it needs to either read it's voxel data from disk or generate new voxel data. Generating new data is typically more expensive than loading it from file, though that will depend on your specific implementation of a terrain generation algorithm. Data loading time for individual chunks can be sped up by using smaller chunk sizes (however, smaller chunk size means more chunks are needed to cover the same area, which reduces overall performance due to chunk overhead), and total loading time can be reduced by using a smaller view distance.

Chunk loading speed is automatically balanced for maintaining a steady framerate during loading. You can set the target framerate in the performance settings. Setting a low target framerate will make chunks load faster.

Building chunk meshes
Building a chunk mesh is very expensive and is generally the most expensive operation performed by the engine. Each newly spawned chunk needs to build it's mesh before it can be rendered, and a chunk mesh will also need to be rebuilt every time a block in that chunk is changed. Block changes can also necessitate the rebuilding of one or more neighboring chunks, all of which have to be performed within a single frame. It's therefore very important to keep the chunk size low enough to allow at least four or five mesh updates to happen within a single frame without causing a significant performance drop. The cost of a mesh update is directly related to the size and complexity of the mesh. A chunk with many exposed block faces, for example, will be more expensive to update than one with only some relatively flat and simple terrain.

The chunk's collider is built simultaneously with the mesh. Disabling the generation of colldiers (Engine Settings: Generate colliders) will significantly improve mesh building performance, though at the cost of disabling all built-in collision detection.

Note: When building the mesh, the vertex count can in some extreme cases exceed the Unity's vertex limit (~64k). In such cases the mesh will be automatically split into multiple meshes, adding some additional performance overhead. Generally, this will only happen with very complex meshes where most of the block faces are fully visible.

GPU load
On computers equipped with a modern gaming graphics card, the GPU performance should not pose any problems. You will run into CPU bottlenecks way before you hit the graphics card's limits. If you absolutely must optimize here, you can reduce draw calls by using larger chunks, at the cost of higher CPU usage. Smaller view distance and vertex count of the chunks' meshes will also improve GPU performance.

6. Multiplayer

Uniblocks comes with a multiplayer solution which enables you to perform some basic voxel-related multiplayer functions, such as streaming the voxel data from a server and synchronizing voxel changes between clients. Note that Uniblocks does not have any gameplay specific network features (synchronizing player positions, etc).

Uniblocks Multiplayer uses the default Unity networking system for simplicity. In cases where you want to implement a third-party networking system, the scripting overview in this section should help you with converting the code to your target platform.

The Uniblocks server can behave as a client in addition to being a server. All client-side functionality can be used on the server as well, including rendering terrain meshes, editing blocks, etc. In order to conserve resources you can disable rendering meshes on the server through the "Generate meshes" setting in Engine Settings for the server.

The clients and the server communicate using RPCs, which are called through the networkView component of the UniblocksNetwork prefab. This prefab has to be instantiated using Network.Instantiate by the server. By default this is done automatically through the ConnectionInitializer script.

ConnectionInitializer

ConnectionInitializer is a simple script for easily connecting to a server or starting a server. It also automatically instantiates the required UniblocksNetwork prefab on server initialization. By default this script is attached to the Engine prefab.

To use it, simply enable the appropriate setting (either ConnectToServerOnAwake or StartServerOnAwake). The connection will be established using the settings supplied in the other properties of the component.

You can also use the ConnectToServer() or StartServer() functions manually, or alternatively use your own client/server solution. In the case of a custom client/server solution, remember to Network.Instantiate the UniblocksNetwork prefab.

Sending voxel data

When multiplayer is enabled, client-side chunks will not load or generate voxel data, but will instead download the data from the server. The server will store, load and generate new voxel data as it's needed.

In order to send a chunk's voxel data, the server needs to first load that chunk. A chunk will automatically be loaded when a player requests it's data, which means that you generally don't need to worry about spawning chunks yourself on the server.

In order to conserve the server's resources, it's recommended to enable the Engine.ChunkTimeout setting. ChunkTimeout simply causes chunks spawned through ChunkManager.SpawnChunk that have not been accessed for some time to despawn (after saving their data). Any access by clients will reset the timeout timer, so chunks which are actively being interacted with by the players will remain loaded.

Synchronizing voxel changes

Changes performed by a client are sent to the server, which then redistributes that change to all other connected clients. This happens automatically whenever the client uses any of the following functions:
Voxel.DestroyBlock
Voxel.PlaceBlock
Voxel.ChangeBlock

Each of these will also send an OnBlockDestroy, OnBlockPlace, or OnBlockChange event to each client (and the server), as well as additional multiplayer-only events (see: Multiplayer voxel events).

You should use these functions whenever you want to perform any multiplayer block changes. Using Chunk.SetVoxel will only change the voxel locally, without sending the change to other players.

Multiplayer voxel events

Whenever a voxel change is sent to clients, voxel events (OnBlockPlace, OnBlockDestroy, and OnBlockChange) are sent to every client, as in singleplayer. In addition, a multiplayer version of these events is also sent (OnBlockPlaceMultiplayer, OnBlockDestroyMultiplayer, OnBlockChangeMultiplayer).

Through this multiplayer version you can access the NetworkPlayer that originally performed the voxel change (stored in the sender variable). You can also use it to separate multiplayer logic from singleplayer logic, since the multiplayer events are not sent in singleplayer.

Player position tracking

In order to conserve bandwidth, the server can optionally track each player's position as well as their local chunk spawn distance setting (which equates to the part of the world that player has loaded on their machine), and only send voxel updates to those players who are close enough to actually see the change. These are stored in UniblocksServer's PlayerPositions and PlayerChunkSpawnDistances arrays.

If Engine.TrackPlayerPosition is enabled on both the client and the server, the client can send a position update to the server. The position is stored as a chunk index, which means that position updates only need to be sent when the player enters a new chunk. In most cases, you'll want to send a position update to the server every time you call ChunkManager.SpawnChunks.

By default, the ChunkSpawner script will send client's position updates to the server automatically, if Engine.TrackPlayerPosition is enabled. You can also do this manually using UniblocksClient.UpdatePlayerPosition(x,y,z).

If the "Track player position" setting is disabled on the server, the server will always send voxel changes to all connected players. In cases where a player doesn't have the appropriate chunk spawned on their machine (the change is too far from the player to see), the client will simply ignore the update.

Scripting

Below is an overview of the network architecture. For detailed explanations of each network function, see the Scripting Reference.

Sending voxel data:
- Client's Chunk sends a “RequestVoxelData” RPC to the server. This happens whenever the chunk would generate or load it's voxel data in singleplayer.
- Server finds or spawns the requested chunk, compresses it's data, and sends a “ReceiveVoxelData” RPC to the client containing the compressed data as a byte array.
- Client's UniblocksClient receives the RPC and loads the voxel data into the chunk.

Sending voxel changes:
- Client sends a “ServerPlaceBlock” or “ServerChangeBlock” RPC to the server whenever Voxel.ChangeBlock, Voxel.DestroyBlock, or Voxel.PlaceBlock is called.
- Server checks which players need to receive the update (if TrackPlayerPosition is enabled)
- Server applies the voxel change locally (since it can't send RPCs to itself)
- Server sends the voxel change to all other clients through the "ReceivePlaceBlock" or "ReceiveChangeBlock" RPC.
- Client's UniblocksClient receives the RPC and applies the change locally, and also generates the appropriate voxel events.

7. Frequently Asked Questions & Common Issues

How to change the size of the blocks?
You can simply scale the chunk prefab through it's Transform.

How to get rid of grid lines showing up on blocks?
These are usually caused by anti-aliasing. To get rid of them, disable anti-aliasing, or force adaptive anti-aliasing.

The world doesn't save between sessions.
By default, the Debugger autosaves the world every 60 seconds. You can also press V on your keyboard to save the world. If you don't have the Debugger component in your scene, these will not be active, so you'll have to save the world manually, using Engine.SaveWorld().
Also, make sure the Engine Setting "Load/save voxel data" is enabled.

Blocks are not spawning.
Make sure the "Generate meshes" setting in Engine Settings is enabled. Also, in order for chunks to actually spawn, you need some way of letting the engine know where to spawn them. By default this is done through the ChunkLoader script. If you don't have it your scene, you will need to spawn chunks manually through ChunkManager.SpawnChunks(x,y,z).

How to set textures for blocks with custom meshes?
A custom mesh uses it's own UVs to determine it's texture. The UVs can be set in a 3D modelling application, and should be unwrapped onto the texture sheet image. Texture settings in the Block Editor have no effect on blocks with a custom mesh.

8. Quick reference

Getting the ID of a specific voxel
```
ushort Chunk.GetVoxel (Index index)
ushort Chunk.GetVoxel (int x, int y, int z)
```

Placing/changing/destroying blocks (triggering mesh update and voxel events and sending the change to other players in multiplayer - use for gameplay events)
```
Voxel.PlaceBlock (x,y,z, ushort data)
Voxel.ChangeBlock (x,y,z, ushort data)
Voxel.DestroyBlock (x,y,z)
```

Changing a specific voxel (use for direct access to memory)
```
Chunk.SetVoxel (x,y,z, ushort data, bool rebuildMesh)
VoxelInfo.SetVoxel (ushort data, bool rebuildMesh)
```

Iterating through all voxels in a chunk
```
foreach (ushort voxel in chunk.VoxelData)
```

Iterating through all chunks
```
foreach (GameObject chunk in GameObject.FindGameObjectsWithTag
(“UBChunk”))
```

Getting the properties of a specific voxel
```
Voxel Engine.GetVoxelType (ushort voxelId)
```

Getting the prefab of a specific voxel
```
GameObject Engine.GetVoxelPrefab (ushort voxelId)
```

Converting a position into a chunk
```
GameObject Engine.PositionToChunk (Vector3 position)
```

Converting a position into a voxel
```
VoxelInfo Engine.PositionToVoxelInfo (Vector3 position)
```

Converting a voxel into a position
```
Vector3 Engine.VoxelInfoToPosition (VoxelInfo voxelInfo)
```

Changing world name at runtime
```
Engine.SetWorldName(string worldName)
```

Saving the world
```
Engine.SaveWorld()
Engine.SaveWorldInstant()
```