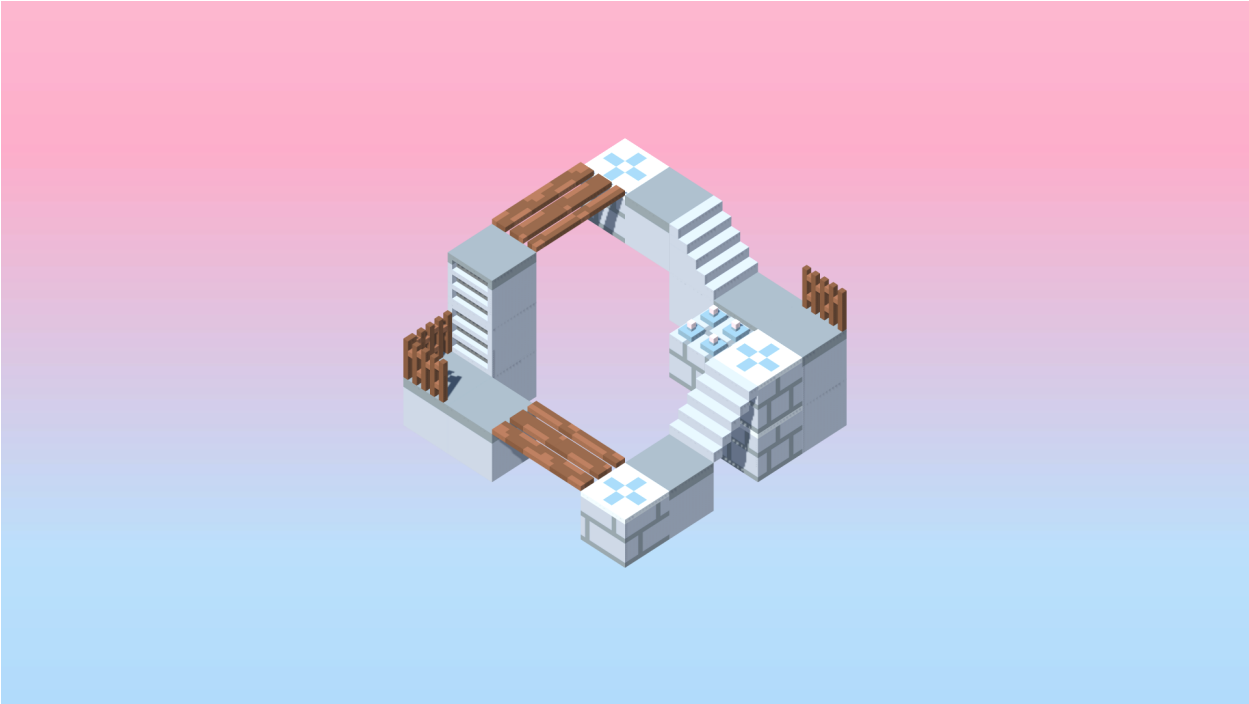
# Autotiles 3D

## Grid Level Editor and Auto-Tiler



Contact: philipbeaucamp.inquiries@gmail.com
Website: sites.google.com/view/philipbeaucamp
Unity Asset Store:http://u3d.as/2GsL

**Table of contents:**

# 1. Introduction

Autotiles 3D is a grid level editor that allows the user to quickly build levels on a three dimensional grid with built-in auto-tiling functionality.
Tilesets and auto-tiling are commonly used concepts in traditional 2D game development. This tool aims to bring this functionality to the three dimensional space, by working with meshes and prefabs, instead of textures, as tiles.

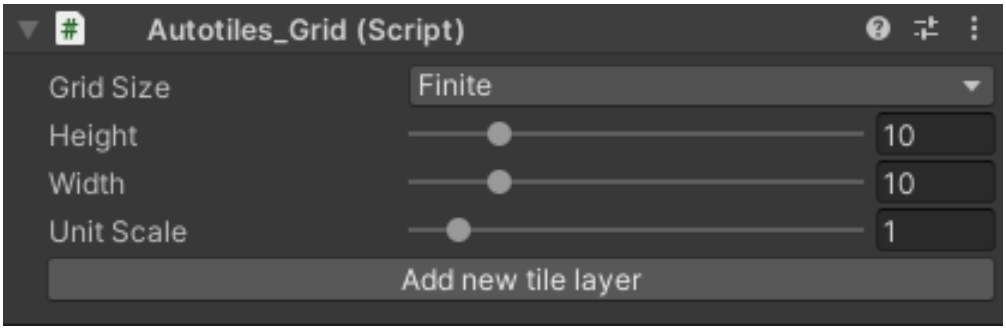
# 2. Quick Overview

The three main components that make up this tool are grids, layers and tiles. To start adding layers and tiles to a grid, you first need to place the grid component **Autotiles3D_Grid** on a gameobject. Then an arbitrary amount of **Autotiles3D_TileLayer** components can be added as children of the grid. While the grid defines the actual layout, like size and spacing, the individual tiles are held in their respective layers.
The **Autotiles3D_TileGroup** scriptable object which can be created via the settings menu (top menu: “Tools/Autotiles3D/Settings) or by right clicking in the project (Create/Autotiles3D/TileGroup) holds a set of **Autotiles3D_Tile**s which can then be placed in the **Editor Mode**. The Editor Mode for a layer is automatically activated when the tile layer is selected in the hierarchy.
Please read through 3.4 Editor Mode & Controls to get an understanding of the controls.

# 3. User Manual

## 3.1 Grids

The **Autotiles3D_Grid** component is needed to define the three dimensional grid layout which an **Autotile3D_TileLayer** uses to place its tiles. To make use of the grid layout, simply place a layer as a child of the grid object.
Even though we use the term *tile layer* which might suggest a two dimensional plane, in this context a layer holds a list of tiles placed in a three dimensional space. A layer placed under a 10x10x10 grid can therefore hold up to 1000 tiles.

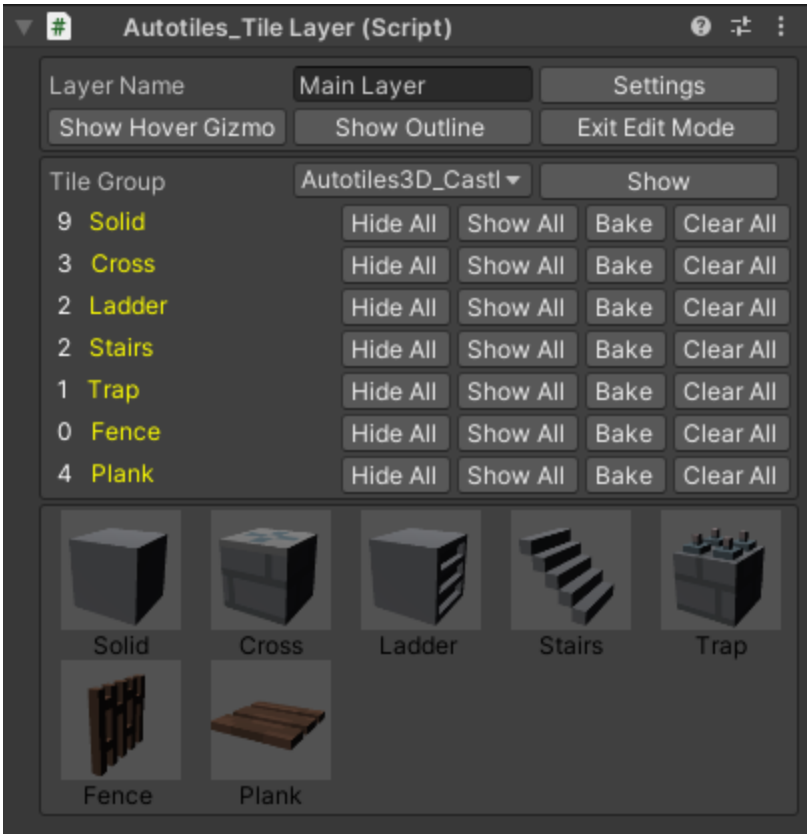



The **Grid Size** can be set to either a finite or infinite grid size. If finite, tiles can only be placed inside the grid boundaries, defined by the **Height** and **Width** parameter. The **Unit Scale** determines the size of a grid cell. Pressing the **Add new tile layer** button automatically adds a new child GameObject with the layer component on it.
All grid options can be changed dynamically even after tiles have been placed.

Remark: The grid's transform can be moved and rotated, but not scaled. Use the **Unit Scale** to increase the grid cell size.

## 3.2 Layers

An **Autotiles3D_TileLayer** holds all the individual tiles. After selecting an **Autotiles3D_TileGroup** from the dropdown menu, the tile group's tiles can be selected by either clicking on the thumbnail or pressing the respective top row hotkey 1-9.
Hovering the mouse in the scene window (inside the grid boundaries) will preview the current tile, which can be placed or removed by using the left or right mouse button. Visibility of the tiles can be toggled by pressing **Hide All** or **Show All** in the inspector. **Clear All** will delete all of the chosen type's tiles in the current layer. Changes to the tile groups can be made via **Settings** or **Show** which selects the current tilegroup in the project. Clicking **Bake** combines all meshes of the respective tile (details under 3.5 Baking)




Remark: All tile layers work independently and are oblivious to each other. This means that even though two layers might be placed below the same grid, the auto-tiling functionality and placeability is only determined by the currently selected layer.

Remark: If Gizmos are disabled you can't interact with the tool in the scene.

Update 1.1 Added a "Refresh" button that manually forces a tile rule check for every placed tile.

## 3.3 TileGroups, Tiles and Rules

To start placing tiles, they first have to be defined on a scriptable object called **Autotiles3D_TileGroup**. The scriptable object can be created by right clicking in the project view and selecting "Create/Autotiles3D/TileGroup" or by using the settings menu.
Each tile needs a *unique* name and a *default* GameObject. The default object is what will be placed when no rules for the tile are defined or when none of the available rules apply.
Autotiles3D works with both prefabs as well as fbx files but no scene game objects can be chosen for a tile.
To make use of the auto-tiling functionality, a set of rules per tile can be defined. If a rule's conditions are all met, the object specified by the rule will be placed instead of the tile's default one.

Each tile in a three dimensional space can have 26 neighbors (e.g. “above”, “in front”, “down and to the right”. For each neighbor, one of three conditions can be specified:

 Neighbor is ignored and not taken into the rule’s calculation

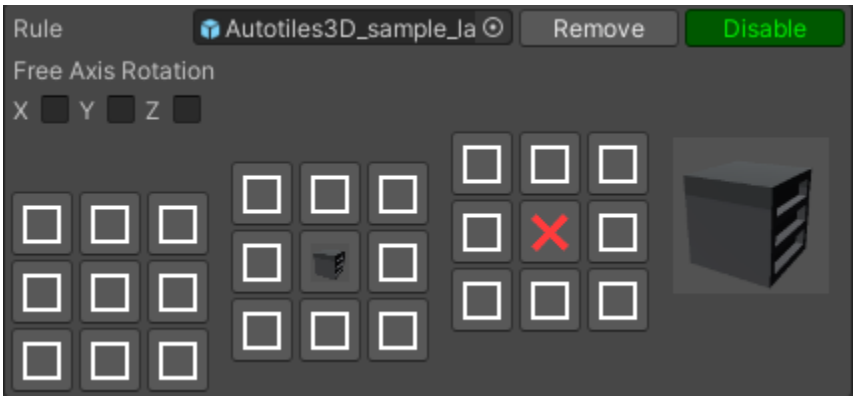
 Neighbor is required. If missing, the rule will fail.

 Neighbor has to be missing, else the rule will fail.

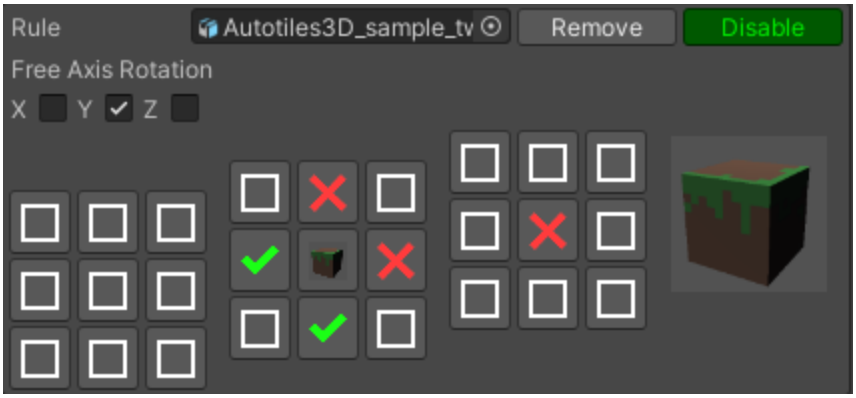
To set the conditions in the inspector, the neighbors “**below”,** on the **“same height”** and **“above”** have been split into three 3x3 selection grids, seen from a **top-down perspective**. To set any conditions of any *below* neighbors, use the left 3x3 grid. Likewise, to set any conditions for the neighbors of the *same height* or *above*, choose the middle or right grid.

Remark: All rule calculations are being performed in the **local space** of the game object. To confirm the forward, up and right directions of a tile, it is recommended to toggle the **Show Hover Gizmo** button found in the tile layer inspector.

To illustrate the above, here are two examples:



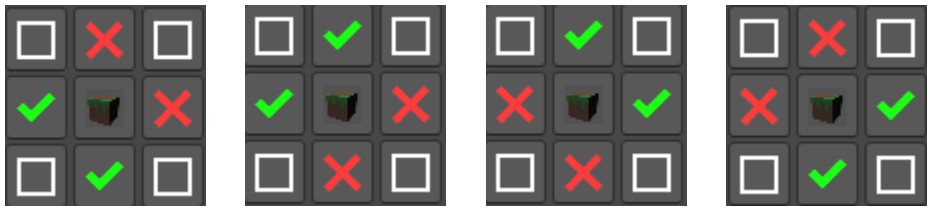

This rule passes whenever there is no tile placed directly above it.




This rule passes when all of the following conditions are true:

- There is no tile directly above it
- There are no tiles in front or to the right (same height)
- There is one tile to the left and one behind it (same height)

Additionally a rule can be set to freely rotate around its own local axis by toggling one or more of the **Free Axis Rotation** fields. As can be seen above in the second example, the rule is allowed to rotate around it's local Y-Axis. This means that the rule is being checked four times, with an added 90 rotation around the local Y-Axis. This is effectively the same as creating four different tile rules as above, with the middle grid, respectively, being set to



Using this functionality can greatly reduce the amount of rules.

Remark: Multiple **Free Axis Rotation** can be selected, however the rotations are all being checked individually given the default rotation, meaning it won't check all four axes rotation for each other possible Axis Rotation. If all three axes are selected, the rule will first rotate four times around it's local X-Axis and if no rotation passes the condition, it will next try to rotate four times around the local Y-Axis and so on.

Remark: The order of rules matters. If there are two or more rules whose conditions are met, the first one will be selected. For performance it is recommended to place the most common rules at the top of the list.

## 3.4 Editor Mode & Controls

Tiles can be placed, modified and removed in the scene view when entering the Edit Mode of a tile layer. Clicking on any gameobject with a **Autotiles3D_TileLayer** component automatically activates the edit mode.
It is difficult to navigate the edit mode without the use of hotkeys and many functions are only available via hotkeys, so it is highly recommended to start off by reading through the below listed controls.

| | |
|---|---|
| Left Mouse Button (click or drag) | Place tile |
| Right Mouse Button (click or drag) | Remove tile |
| Alt + Mouse Wheel (scroll) | Move placing layer up/down |
| Shift + Mouse Wheel (click on block) | Move placing layer to block height |
| Shift + Left Mouse Button (click on block and drag and pull) | Extrude tile(s) in pulling direction |
| Shift + Right Mouse Button (click on block and drag and push) | Remove tile(s) in pushing direction |
| Shift + Mouse Wheel (scroll) | Switch between extrusion/removal of a single tile or the tile including all its adjacent neighbors. |
| Left Control + Mouse Wheel (scroll) | Rotate the current hovering tile by 90 degrees clockwise around the Y-Axis |
| Top Row Keys 1-9 (while scene focused) | Quickselect tile 1-9 of current tile group |
| Escape | Exit Edit Mode |

## 3.5 Baking

Tiles of the same type in a layer can be baked together, which will automatically combine all meshes by material. It is always possible to **rebake** after adding new tiles of the same type too.

Remark: Baking tiles does *not* remove the original tile or mesh. It creates the new combined mesh and then merely disables the “View” target specified in the **Autotiles3D_BlockBehaviour** component. This means that even after deleting the baked mesh, you can view all the original tiles by pressing **Show All** in the inspector of the grid layer.

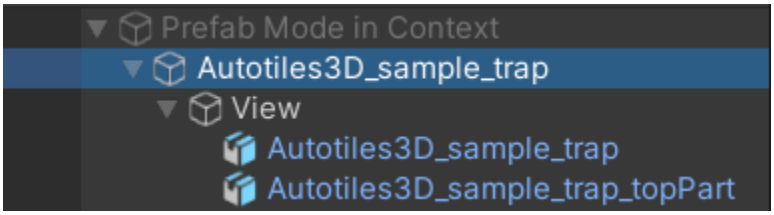
Baked tiles will not be removed upon clicking **Clear All**. If they really need to be deleted one can remove them manually from the hierarchy (while keeping the baked parent). In this case the tiles are truly removed from the layer and don't exist in the system anymore. Consequently, rebaking will not be able to re-combine the now deleted meshes again.

Combined meshes are saved as assets under Assets/Autotiles3D/Content/CombinedMeshes.

## 3.6 The BlockBehaviour Component

The **Autotiles3D_BlockBehaviour** is a MonoBehaviour component that needs to be present on all tiles. If it is missing (e.g. the user is working with .fbx models) it will be automatically added onto the instantiated game object that will be created when a tile is placed.

Each BlockBehaviour has a **View** field of type GameObject which should represent the root of the model (not necessarily the root of the prefab itself). It is the target of this field which is used to toggle the visibility of the tiles as well as to find the mesh filters for baking.
The target is allowed to be the prefab/fbx root itself too, however in this case toggling the view could also turn off any scripts and components of interest on the GameObject itself which might not be desired. For an optimal overflow it is generally recommended to separate the prefab root and script logic from the meshes. When working with multiple mesh filters, the mesh objects should all be placed on or below the **View** game object, as can be seen below.

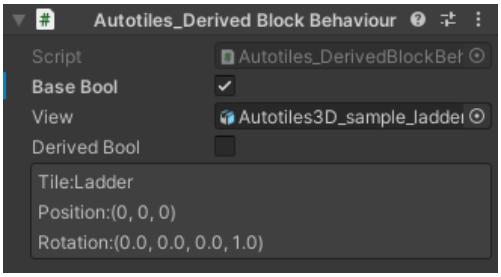



The BlockBehaviour holds data such as the grid position/rotation or info about the tile, however the main functionality of the MonoBehaviour is that if an instance needs to be re-instantiated because of tile rule changes, the data inside the old BlockBehaviour gets copied and transferred to the BlockBehaviour of the new instance if their class types are matching.
More specifically, if the new BlockBehaviour is a derived class of the old BlockBehaviour any matching fields get copied and the data put from into the new BlockBehaviour. The same happens for matching fields if the new BlockBehaviour is the base class of the old BlockBehaviour.

The following illustration gives an example of the above:

Because of a neighboring tile addition in grid position (0,0,1), the tile placed in (0,0,0) needs to change its instance from Prefab A to Prefab B to respect the rules defined for the tile. Instance A of Prefab A held the base class **Autotiles_BlockBehaviour** whereas the new Prefab B to be

instantiated holds the derived BlockBehaviour class **Autotiles_BlockBehaviourDerived** which inherits from the former.


Since both classes share the field “BaseBool” and Instance A’s “BaseBool” was set to true, the data gets copied across the derived class so that “BaseBool” of the newly created class on Instance B is true as well.

# 4. Exposed API

First and foremost, Autotiles 3D is designed as an editor-only, non-runtime tool to use within the Scene view of the Unity Editor. Thus follows the following

**Disclaimer:** It should be understood that manually manipulating the internal nodes stored inside the tile layers via code comes with its own risks and can lead to bugs or data loss. Please proceed at your own risk!

If you nevertheless want to interact with the API, make changes to it or have some access to some of the more useful functions, the following methods are the safest way to interact with the code. The actual code is encapsulated by a “Exposed API” region inside the c# files and also commented.

**Autotiles3D_Grid:**

- **bool IsExceedingLevelGrid(Vector3 internalPosition)**
  Is the internalPosition exceeding the boundaries of the grid

- **List<Autotiles3D_BlockBehaviour> GetBlocks(Vector3Int internalPosition)**
  Returns a list of all the BlockBehaviours at the internalPosition. Can return multiple blocks if working with multiple layers

- **Vector3 ToWorldPoint(Vector3 internalPosition)**
  Grid to world position

- **Vector3 ToWorldDirection(Vector3 internalDirection)**
  Grid to world direction

**Autotiles3D_TileLayer:**

The most critical data inside the TileLayer is the "Dictionary<Vector3Int, InternalNode> InternalNodes" dictionary which stores the information of every internal node inside the layer. The internal node is the class that holds information about the tile,layer,block behaviour, prefab, position, rotation and more. Reading from the internal node data is safe. Writing data to internal nodes manually is not recommended and should only be done via the following exposed functions.

- **void TryPlacementSingle(Vector3Int internalPosition, Quaternion localRotation, Autotiles3D_Tile tile = null)**
  Adds a single node to the layer

- **void TryPlacementMany(List<Vector3Int> localPositions, List<Quaternion> localRotations, List<Autotiles3D_Tile> tiles)**
  Adds multiple nodes to the layer

- **void TryUnplacingSingle(Vector3Int internalPosition)**
  Removes a single node from the layer

- **void TryUnplacingMany(List<Vector3Int> internalPosition, bool waitForDestroy = false)**
  Removes multiple nodes together (better performance than removing one by one)

- **void RefreshNode(InternalNode node)**
  Refreshes a single node which checks the neighboring conditions anew and updates the gameobject instance accordingly.

- **void RefreshAll(Autotiles3D_Anchor anchor, bool forceImmediateRefresh = false)**
  Refreshes all nodes of the corresponding anchor.

Please understand that any TryPlacing/TryUnplacing/RefreshNode calls only update the internal data and require a final call to the function “public void Refresh()” to translate any changes made to the layer’s internal nodes to the scene.
This is done for performance reasons, so that multiple changes of intern nodes (**fast)** can be “stacked” internally until the next call of Refresh() which updates the scene by instantiating the correct gameobjects etc (**slow)** at once. Refresh() is called **once every OnSceneGUI** update.

**Autotiles3D_MeshCombiner**

- **bool CombineMeshes(List<GameObject> roots, ref GameObject newParent, string path)**
  Bakes meshes found in the specified gameobjects children

# 5. Currently known issues & limitations

- Since the Undo API's functions (eg. Undo.RegisterCreatedObjectUndo and Undo.DestroyObjectImmediate) are very expensive functions, the editor can get extremely slow when trying to instantiate multiple hundreds of prefabs at the same time. This shouldn't be a problem for most use cases, but users who need to work with very large tile extrusions have the option to turn off any Undo registrations in the project settings which greatly improves performance.

Inquiries/Bugs/Feature Requests: philipbeaucamp.inquiries@gmail.com