Building Block Editor.

doc v 1.1

Thanks for trying out Building Block Editor from Bumpkin Labs. We hope you find it enjoyable to use and it adds flexibility to your game design process.

What is it?

Building Block Editor is a framework for creating an in-game level editor that is simple to use, both for the development team and, more importantly, the player.

The basic principle is you create a set of building blocks that can be linked together with connectors. We like to think of each object as a 'toy'. These building blocks can be anything you can imagine. Like simple minecraft-esque cubes, complex shapes, enemies with AI data - in fact anything you can dream up!

Contact

Stay in touch with us for further help and info on Bumpkin Labs tools and Bumpkin Bros games.

email: contact@bumpkinbrothers.com
facebook: https://www.facebook.com/bumpkinbrothers
twitter: @bumpkinRich and @bumpkinAndy

You can find more help on our YouTube channel
https://youtu.be/7CWOGMvlDnA?list=PLhOBdAuDweKO8DQ_qT_Ok0CnhKFg1WYMX

Basic Tutorial

Setting the scenes.

Getting started with Building Block Editor is easy. You just need to tweak 1 project setting, 2 scenes (Level Editor and Level Player) and with a couple of menu clicks you'll be up a running!

Set the connector layer.

We need to set aside a layer for the Building Block Editor to use. By default this is layer 8 but you can change this.

1. Open the layers editor by clicking the layers dropdown at the top right of the Unity window and selecting edit layers.

2. Give the layer 8 a name like Building Block Editor, so you know what it's being used for. If you already use layer 8 (or just want to choose a different layer) than rename the layer you want to use.

3. If you have chosen another layer than 8, open the LevelEditorGloabls.cs script (BumpkinLabs\BuildingBlockEditor\Managers\LevelEditorGlobals.cs) and edit the int BuildingBlockConnector to the layer you have used. Also open the DetailedMeshCollider class (BumpkinLabs\BuildingBlockEditor\Scripts\DetailedMeshCollider.cs) and change scanLayer.

Creating the playing scene (this is where you will play the game)

1. Create a new scene in your project, save it as "Level Player".

2. Add your new level to the Build Settings (File -> Build Settings -> Add Current)

3. Add a Level Loader (Tools -> Bumpkin Labs -> IGE -> Add Scene Loader

Creating the editor scene (this is where you will create / edit levels)

1. Create a new scene in your project, save it as "Level Editor."

2. Add your new level to the Build Settings (File -> Build Settings -> Add Current)

3. Set the scene up as a level editor by using the build in converter (Tools -> Bumpkin Labs -> IGE -> Convert Scene To Level Editor)

Now you have a basic editor scene containing the default basic block. If you press play you'll see your cube which you can orbit around by holding the middle mouse button.

Adding items

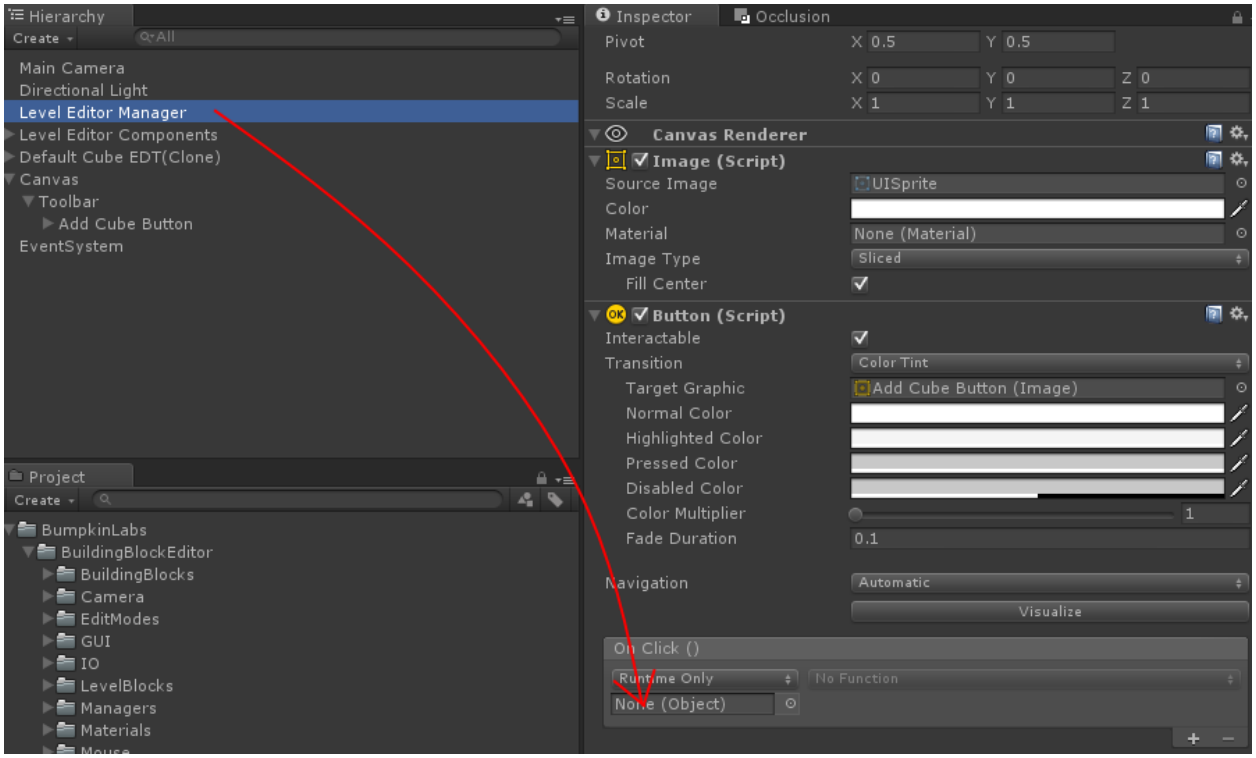
Let's add functionality to build more cubes.

You will usually interact with Building Block Editor via the LevelEditorManager script. We're going to call the LevelEditorManager.EnterAddBlockMode(BuildingBlock block) function passing in the default cube prefab that comes with the project.

1. Create a button in your scene and name it 'Add Cube' (find out more about the Unity UI here - http://docs.unity3d.com/Manual/UISystem.html)

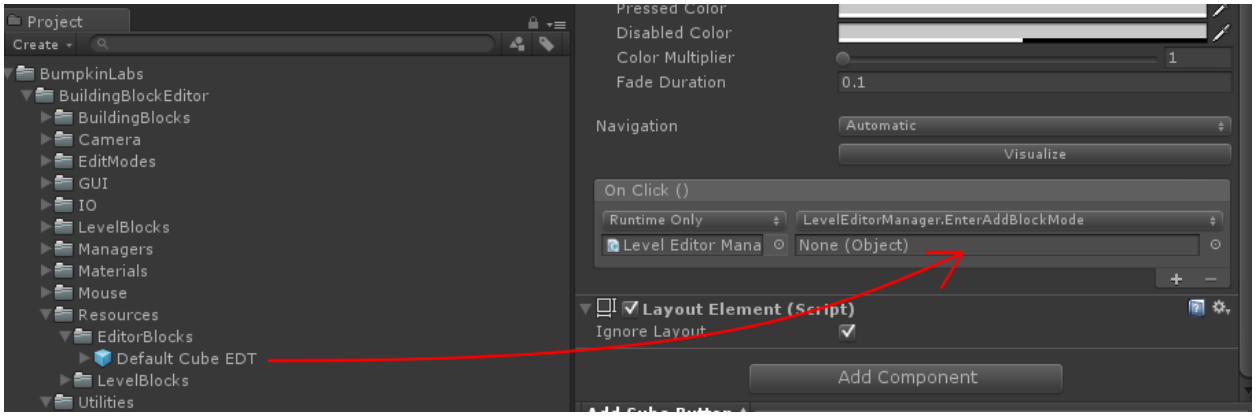
2. Add an OnClick event to the button.

3. Locate the Level Editor Manager game object in the scene hierarchy and assign it to the OnClick event you just created.



4. Choose the LevelEditorManager.EnterAddBlockMode function.

5. For the object parameter we need to pass in a BuildingBlock prefab. Use the default cube that comes with the project. Located under BumpkinLabs -> BuildingBlockEditor -> Resources -> EditorBlocks -> Default Cube EDT



Now you can run your project, click on the 'Add Cube' button and you can start adding cubes by left clicking on the existing cube.

Creating Building Blocks

Building blocks are the main foundation of using Building Block Editor. They come in pairs, the BuilldingBlock which is used in the editor, and LevelBlock, which is used in game. So when a player creates a building block in the editor and runs the level, the level block equivalent is created. The reason they're two separate items is you may want them the behave completely differently. For example an enemy spawn point may appear as an arrow or X in the level editor while in the game you just use the position as a starting point for a spawning script.

Luckily they're easy to make. We'll create another cube with a red material.

1. Create a cube (GameObject -> 3D Object -> Cube) and rename it "Red Cube", place it somewhere in your scene where you can see it.

2. Create a new material, set the color to red and apply it the the mesh renderer of the cube.

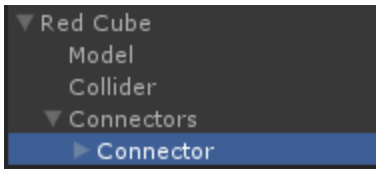
3. With the cube selected we'll convert it to a building block using the conversion tool - Tools -> Bumpkin Labs -> IGE -> Convert To Building Block.

Your cube is now a building block. All you need to do is set up the connectors.

Connectors tell the editor how new building blocks can be attached to building blocks in the scene, letting the player create levels by snapping the connectors together.

We'll set the cube up to have connectors on each side.

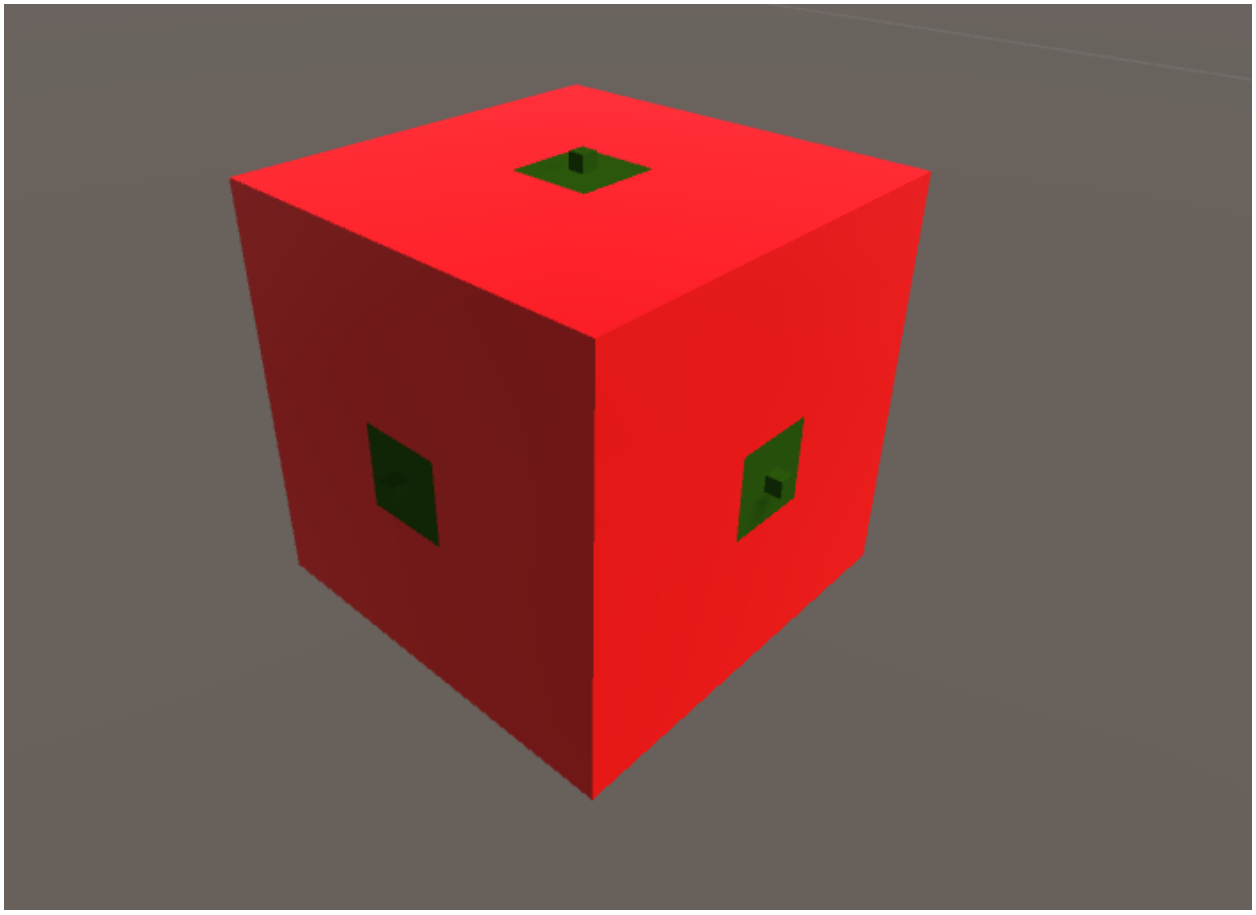
1. When you use the 'Convert to building block' tool it creates a connector for you, expand the building block in the hierarchy to see its children, and expand the connectors node to see it.



2. With the connector game object selected you now need to position it on one of the sides of the cube. Click the 'pick point' button in the property inspector and then left click on one of the sides of your cube while holding shift. This places the connector at this spot, and snaps the position to the nearest 0.5 of a unit.

3. Create 5 more connectors. With the connector still selected hit CTRL + D to duplicate your select 5 times.

4. Move through the connectors using the Pick Point tool to place a connector on each side of the cube.



Saving the prefab.

Now we need to create the prefab our editor can recreate for us. We need to save the prefab in a specific folder structure; Resources / EditorBlocks. You can place the resources folder anywhere in your project you like, and have multiple folders but all building blocks must be stored using this pattern. For the example I created a folder called Building Blocks / Resources / EditorBlocks and dragged the ‘Red Cube’ game object from my Hierarchy window to the EditorBlocks folder in my project window.

Creating a Level Block.
Level Blocks are the version of a Building Block that gets created when we load our level. They can be totally different to the building block - for example a character spawn point may appear as a token in your level editor but spawn a character in the level.

Let’s turn our Red Cube game object into a Level Block.

All we need to do for this is to select the Red Cube game object and click on Tools -> Bumpkin Labs -> IGE -> Convert BuildingBlock to LevelBlock. This will convert the game object into a stripped down version of a building block.

We also need to save our Level Block as a prefab. Like Building Blocks Level Block prefabs have to be saved in a specific folder; Resources / LevelBlocks. Follow the same steps to create a Building Blocks / Resources / LevelBlocks folder and drag the Red Cube in to create a prefab.

You can now delete the Red Cube game object from your scene, as you have it saved as a prefab.

Now, duplicate the Add Cube button you created earlier to create a 'Add Red Cube' button. Change the argument for the EnterAddBlockMode to the Red Cube prefab in the Building Blocks / Resources / EditorBlocks folder.

Playing the game.

To complete the tutorial lets add a final button to your scene to allow you to change to the playing scene.

1. Duplicate the Add Red Cube button and name the new button 'Play Level'

2. Change the function being called by the OnClickEvent from EnterAddBlockMode to TestLevel. For the argument enter 'Level Player' - which should be the name of the scene you created earlier.

Now you can play the editor, create some new blocks (red and white) and then press the play button to view your level in the 'Play Scene'. Of course the play scene won't do anything yet but it should show the blocks are being created showing they're being saved in the right location.

Making Complex Building Blocks.

Creating scenery is great, but you'll probably want to create some more complicated objects that require user input during level design. A platform that moves between two points for example or a patrol route for an enemy.

As the uses of a building block are limited only by imagination the implementation is left open for the developer. Building Block Editor contains everything you should need to pass what has been setup in the editor to the level.

Editable Blocks

If a building block is editable then when you click on it after calling LevelEditorManager.EnterSelectBlockMode() control is passed to that object and passed back to the editor when you have finished.

To make an Editable Block you need to add a script to the building block that inherits from EditableBlock. This has several overridable methods that you can hook into.

EnterEditMode() - this method will be called when the player clicks on your building block in Select Block Mode and is usually used to show the interface for editing .

ExitEditMode() - this method will be called when a LevelManager changes the edit mode.

PrepareForSave() - this method will be called before the level is saved, its purpose is to allow you to store the metadata for the object.

string MetaData; - the metadata property is used for storing data about the object - it is this data that's passed to the level block.

When a level block is created Unity's BroadcastMessage method is used to call SetMetaData(string metaData) on any scripts that implement it. So to get hold of the metadata on the play side you just need to add a script to the Level Block that implements that method.

Making Decorative Items.

Sometimes you don't want to be restricted in the placement of an item in the scene. You may want to add some trees at any point to break up any uniformity.

The BuildingBlock component and Connector components can handle this.

BuildingBlock.AllowDecorativePlacement - if this is true then connectors will be ignored when this item is being placed. Instead the object will be placed using the normal of the polygon currently selected.

BuildingBlock.AllowDecoratveObjects - if this is true then decorative objects can be placed here. You should switch this off on moving objects.

Saving and Loading

The LevelLoader component is responsible for loading levels. As long as there is one in your scene it will trigger during the OnLevelLoaded event and load the level.

To set the current level that will be loaded you use PersistentSceneData

To load a level from a file you will need to

1. Call PersistentSceneData.Instance.FileName = “full path to your level data”
2. Load the scene containing a LevelLoader.

To load a level from a text asset you will need to

1. Call PersistentSceneData.Instance.TextAsset = textAssetToLoad;
2. Load the scene containing a LevelLoader.

To save an edited level you call LevelEditorManager.Intance.SaveLevel(string fileName) - fileName in this instance will be appended to Application.persistentDataPath. So LevelEditorManager.Instance.SaveLevel(“MyLevels\\Level.txt”) with create a file in C:\Users\USERNAME\AppData\LocalLow\PROJECTCOMPANYNAME\PROJECTNAME\MyLevels\Level.txt

Changing the connectors.

By default all building blocks are created using the prefab located in BumpkinLabs\BuildingBlockEditor\BuildingBlocks\Prefabs if you alter this prefab it will be used in subsequent ‘Convert To Building Block’ calls.

If you want to change all connectors that have already been added you can do this using the ‘Update Connectors’ utility.

1. Select the prefab you want to use in the project window.
2. In the menu choose Tools -> Bumpkin Labs -> IGE -> Update Connectors.

This will look for instances of connectors within prefabs in your project and update them.

If you don’t want certain connectors to be affected by this process use the disableAutomatedUpdate property of the connector.

THIS PROCEDURE IS DESTRUCTIVE AND WILL ALTER PREFABS IN YOUR PROJECT. PLEASE BACKUP YOUR WORK BEFORE RUNNING UPDATE CONNECTORS

Mesh Combining.

It is possible to combine meshes that use the same texture together to form a single mesh and mesh collider.

To do this you need to add a StaticModel component to the LevelBlock, on the object that contains the MeshRenderer and MeshFilter components.

You also need to enable the EnableMeshCombine property of the LevelLoader component in your player scene.

The properties of the StaticModel component are

public string staticModelName = "";
The name of the model - all static models with the same name will be combined (and changed to a single texture if multiple are used.)

public float combineTollerance = 10;
The distance between blocks to link

public bool applyMeshCollider = false;
If true then a mesh collider will be added to the created mesh (and any collider attached to this object will removed)

public bool recalculateNormals = true;
Force the new mesh to recalculate its normals.

public bool destroyLevelBlock = false;
If true then the parent LevelBlock will be destroyed.

Demo Project

Building Block Editor comes with a demo project of a rolling ball game.

To run the demo you need to add the following scenes to the Build Settings

- Assets\BuildingBlockDemo\Scenes\BBExMenu
- Assets\BuildingBlockDemo\Scenes\BBLevelEditor
- Assets\BuildingBlockDemo\Scenes\BBExLevelPlayer

The demo expects the default Unity input settings for Horizontal and Vertical axis to be set.

Run the demo by starting the BBExMenu scene.

Building Block Editor Components Overview.

The following is an overview of the components in Building Block Editor.

LevelEditorManager.cs

This is the main class that controls and gives access to the level editor.

Inspector Properties

BuildingBlock[] initialBlocks;
Your scene needs to contain some initial blocks to build on. These should be specified here so the manager knows they existed in the scene (rather than from saved data)

Material potentialPlacementMaterial;
A temporary material to color an object when it is in a placeable position.

Material potenialPlacementBlockedMaterial;
A temporary material to color an object when it cannot be placed.

Material deletionHighlightMaterial;
A temporary material to color an object when it is being hovered over in delete mode.

Material editHighlightMaterial;
A temporary material to color an object when it is being hovered over in edit mode and the object is editable.

Camera editorCamera;
The main camera being used to do the editing.

bool returnFromTestOnEscape;
If true then pressing escape will return you from the play scene to the editor.

Public Properties;

LevelEditorManager Instance;
A static property giving access to the LevelEditorManager

Functions;

EnterAddBlockMode(Object buildingBlock)
Creates a new instance of the building block supplied in the argument and allows it to be placed.

EnterDeleteBlockMode()
Enters a mode where you can click on building blocks to delete them.

EnterSelectBlockForMovingMode()
Enters a mode where you can click on a block to move it to a new location.

EnterSelectBlockMode()
Enters a mode where you can select editable blocks.

SaveLevel(string levelName)
Saves the level. levelName is combined with Application.persistentDataPath. So myLevel.txt would be saved (on windows) to C:\Users\USERNAME\AppData\LocalLow\PROJECTCOMPANYNAME\PROJECTNAME\myLevel.txt

LoadFile(string fileName, string editorSceneName)
Loads an editor scene.

fileName = a file name, it will be combined with Application.persistentDataPath as in SaveLevel.

editorSceneName = the name of a scene in your project setup as a level editor.

BuildingBlock.cs

All building blocks must contain this component.

string buildingBlockName
The main name of this type of building block. Can be duplicated to find blocks of the same type - for example you may want to call all Cubes ‘Cube’.

string buildingBlockSubName
The sub name for the building block should be unique to the type. So while there can be many Cube types - there should only be one Cube Red type.

DetailedMeshCollider blockMesh;
The DetailedMeshCollider checks to see if your new object is colliding with any building blocks already in the scene.

GameObject blockModel;
The model that represents the building block. This model will have its material altered when being edited.

bool allowDecorativePlacement;
If true then this building block does not need to be placed on a connector.

bool allowDecorateObjects;
If true then it will be possible to place objects with allowDecorativePlacement set to be placed at any location on them.

GameObject[] inSceneObjects;
Store any objects here that should be enabled when the object is placed. For example a ‘off set point’ for a moving platform.

bool forceModelRotation;
If true then the object will rotate to forcedRotation when placed in the scene.

Vector3 forcedRotation;
The rotation the object will rotate to with forceModelRotation is true.

bool locked = false;
If this is set to true then the blocks cannot be deleted from within the editor. Use this if your level requires certain blocks.

bool dontCheckOverlaps = false;

If true then the block is allowed to overlap other objects in the scene when being placed.

BuildingBlockConnector.cs

Connectors control the way blocks are attached / attached to.

public bool useInPlacementMode = true;
If true then this connector can be selected on the ‘incomming’ object.

public bool useInSceneMode = true;
If true then this connector can be selected to place the incomming object onto.

public GameObject highlightObject;
The game object that shows where the connector is. When you hover over a building block the connectors become visible.

public Collider raycastCollider;
This is the collider that the mouse will be over to select this connector. If null then a MeshCollider will be found in children.

public bool allowRoation = true;
If true then the user can rotate the incomming item around this connector by pressing shift + right click.

public bool disableAutomatedUpdate = false;
If true then this connector will not be updated by the ‘Update Connectors’ utility.

LevelBlock.cs

public string buildingBlockName = "";
The main name of this type of building block. Can be duplicated to find blocks of the same type - for example you may want to call all Cubes ‘Cube’. Must match a BuildingBlock of the same type.

public string buildingBlockSubName = "";
The sub name for the building block should be unique to the type. So while there can be many Cube types - there should only be one Cube Red type. Must match a BuildingBlock of the same type.

public GameObject model;
This can be left null but if LevelBlock is required to do anything with the model then it needs to be set.

public bool forceModelRotation;
If true the model object will be rotated to forcedRoation.

public Vector3 forcedRotation;
If forceModelRotation the model object will rotated to this position.

public bool randomRotation = false;
If true the model will rotate randomly (fixed to 90 degrees)

LevelLoader.cs

The LevelLoader component does a few things that you may want to hook into in your game.

When the level is loaded a LoadLevelCompleted message is passed to all scripts in all LevelBlocks.

Vector3 LevelLoader.Instance.Min
The position of the levelblock at the minimum extent of the level.

Vector3 LevelLoader.Instance.Max
The position of the levelblock at the maximum extent of the level.

Vector3.LevelLoader.Instance.Centre
The center point of the level.