New Game Studio

Advanced Culling system manual

Table Of Contents

Contact and Support

If you have any questions regarding the use of ACS or any of my other assets, feel free to contact me through any convenient method:

Asset Store Page : https://assetstore.unity.com/publishers/9290

Discord : https://discord.gg/6EYUn9QhXF

Website : http://thenewgamestudio.com/

YouTube : https://www.youtube.com/@NewGameStud1o

Mail : andre-orsk@yandex.ru

When To Use

Occlusion Culling is a powerful optimization tool that discards objects currently occluded by other objects and not visible to the camera, even if the camera is facing them.

The Advanced Culling System provides solutions for all potential scenarios. You can cull both static and dynamic objects, with or without preprocessing. It supports culling not only MeshRenderers and LODGroups but also lights and any other custom objects.

Please note that culling as an optimization approach is entirely useless in scenarios where no objects overlap in the frame (e.g., top-down shooters).

The Advanced Culling System consists of two modules: Static and Dynamic Culling.

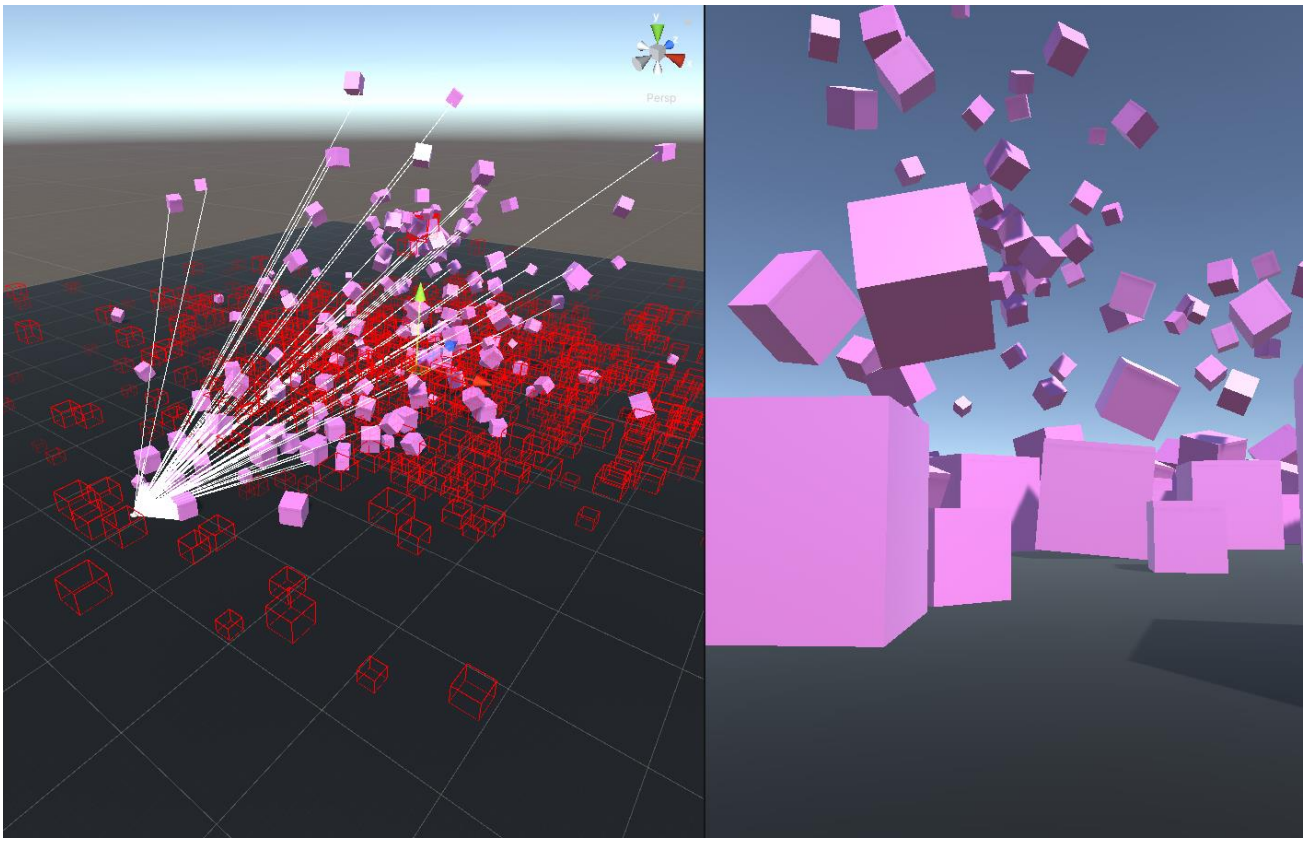
Static Culling Module allows preprocessing the visibility of objects and then simply turning them on and off during runtime. This approach offers deep customization options and generates less load during runtime. I recommend using this approach for small to medium-sized scenes.

Dynamic Culling is a module that simplifies the optimization of both static and dynamic objects without the need for preprocessing. This solution is perfect for large and extremely large scenes.

Idea (Dynamic Culling)

The idea behind **Dynamic Culling** is quite simple. Rays are cast from the camera, distributed across the screen in a specific sequence.

If a ray hits an object, that object is considered visible. If no rays hit the active object within a specified period, the object is considered invisible and is turned off.



Despite the fact that raycasting is a resource-intensive operation, numerous optimizations are used under the hood, including Burst and Jobs.

This approach allows for maximum minimization of CPU load and enables the use of Dynamic Culling without preprocessing for scenes of any size.

Quick Start (Dynamic Culling)

1) Creating a Controller

The first thing you need to do is create an instance of Dynamic Culling. Click on **Tools -> NGSTools -> Advanced Culling System -> Dynamic**.

Now, if you click on the Dynamic Culling instance, you will see a custom interface in the inspector, simplifying the scene setup.

A detailed description of each field can be found in the "Components Description" section.

2) Assigning Cameras

Next, you need to assign the cameras. Click on the "Cameras" dropdown and select **Assign Auto**.

Alternatively, lock the inspector window by clicking the **Lock** icon in the upper right corner. Then, select the cameras you want to add in the hierarchy and click **Add Selected**.

3) Assigning MeshRenderers and LODGroups

To specify which objects the system should cull, you need to attach the **DC_SourceSettings** component to objects with MeshRenderer or LODGroup.

You can do this manually or use the DC_Controller component interface.

Using the buttons from the dropdown menu, add the objects you want to cull. To highlight the objects you have added to the system in the scene, enable the **Draw Gizmos** flag.

4) Custom Objects

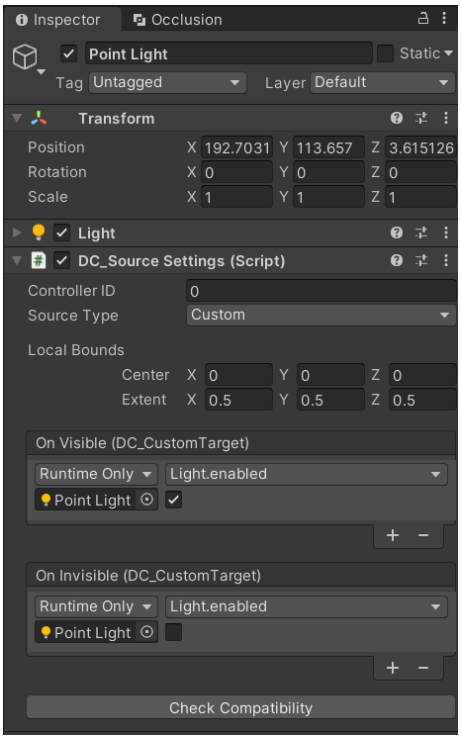
If you want to cull an object that doesn't have a MeshRenderer or LODGroup (e.g., Light or ParticleSystem), or if you want to customize the behavior for onVisible/onInvisible events, use a custom **SourceSettings**.

To do this, attach the **DC_SourceSettings** component to the target object and set the "Source Type" to "Custom."

Then, define the bounds for the object using the special Gizmos in the scene or the "LocalBounds" field.

Next, assign the events that will be triggered when the object is visible or invisible to the camera.

An example setup for a PointLight is shown below:



5) Baking

This step is optional for Dynamic Culling. The main idea is that during the Start method call in **DC_SourceSettings**, some calculations are performed. These calculations are then sent to the **DC_Controller**, which might cause a slight delay when the scene starts. To perform these calculations in the editor for objects already in the scene, you can press **Bake** in the **DC_Controller**. This way, the calculation step during the Start call will be skipped.

How It Works (Dynamic Culling)

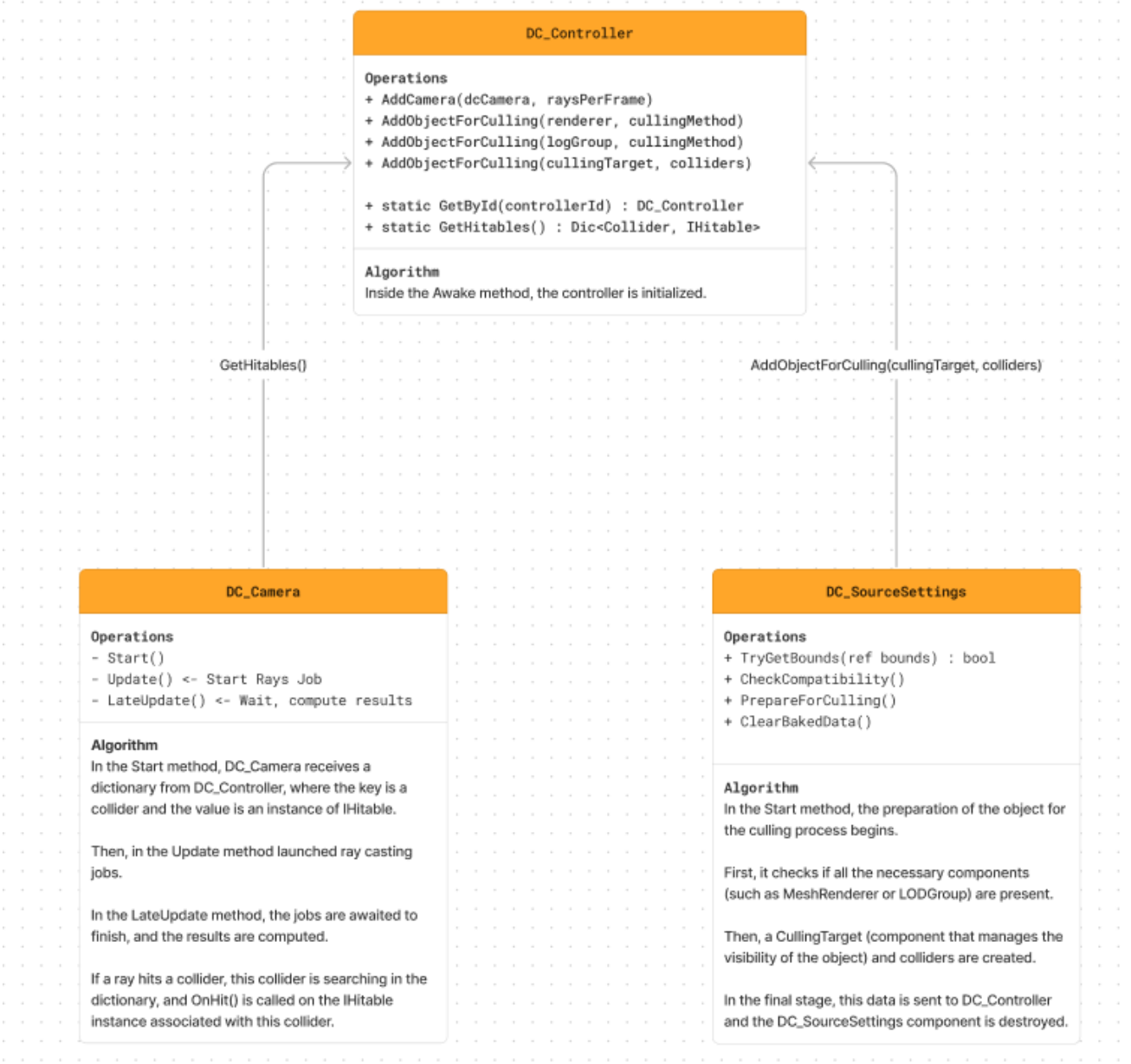
The starting point for the algorithm is the **Awake** method call in **DC_Controller**. In this method, data is prepared, and the controller gets ready to handle requests.

For the **DC_SourceSettings** component, the entry point is the **Start** method call. This method prepares the object for the culling process. First, it checks if the object has all the necessary components (e.g., MeshRenderer). If the required components are present, colliders are created for the object and placed in the **ACSCulling** layer. This layer should appear automatically in Tags And Layers when you first click on **DC_Controller**. It is isolated from interaction with other layers, so it should not affect the current state of your scene.

Next, an instance of **DC_CullingTarget** (this class manages the visibility of the object) is created, and all data is sent to **DC_Controller**. After sending the data, the **DC_SourceSettings** component is destroyed and replaced by the **DC_CullingTargetObserver** component. This component will monitor the state of the GameObject. If the GameObject is destroyed, the Observer will remove its data from **DC_Controller**.

For **DC_Camera**, initialization primarily occurs in the **Start** method. **DC_Camera** receives a dictionary from **DC_Controller** where the key is the collider, and the value is an instance of **IHitable**. Then, in the **Update** method, threads are launched to cast rays. In the **LateUpdate** method, thread completion is awaited, and results are processed. If a ray hits a collider, that collider is found in the dictionary, and **OnHit()** is called on the **IHitable** instance associated with that collider.

A simplified schematic of the algorithm's operation is presented below :


DC_Controller
Operations
+ AddCamera(dcCamera, raysPerFrame)
+ AddObjectForCulling(renderer, cullingMethod)
+ AddObjectForCulling(logGroup, cullingMethod)
+ AddObjectForCulling(cullingTarget, colliders)
+ static GetById(controllerId) : DC_Controller
+ static GetHitables() : Dic<Collider, IHitable>
Algorithm
Inside the Awake method, the controller is initialized.
GetHitables()
AddObjectForCulling(cullingTarget, colliders)
DC_Camera
Operations
- Start()
- Update() <- Start Rays Job
- LateUpdate() <- Wait, compute results
Algorithm
In the Start method, DC_Camera receives a dictionary from DC_Controller, where the key is a collider and the value is an instance of IHitable.
Then, in the Update method launched ray casting jobs.
In the LateUpdate method, the jobs are awaited to finish, and the results are computed.
If a ray hits a collider, this collider is searching in the dictionary, and OnHit() is called on the IHitable instance associated with this collider.
DC_SourceSettings
Operations
+ TryGetBounds(ref bounds) : bool
+ CheckCompatibility()
+ PrepareForCulling()
+ ClearBakedData()
Algorithm
In the Start method, the preparation of the object for the culling process begins.
First, it checks if all the necessary components (such as MeshRenderer or LODGroup) are present.
Then, a CullingTarget (component that manages the visibility of the object) and colliders are created.
In the final stage, this data is sent to DC_Controller and the DC_SourceSettings component is destroyed.

Components Description (Dynamic Culling)

DC_Controller

- DrawGizmos (in play mode, if **MergeInGroups** is enabled): Visualizes the cells into which the scene is divided. Objects within the same cell are considered a group.
- ControllerID: The ID of the controller. All **DC_SourceSettings** with the same **ControllerID** will be processed by this controller.
- ObjectsLifetime: How long an object will be considered visible (in seconds). If no rays hit the object within this time, it is deemed invisible and deactivates.
- MergeInGroups: Whether to group nearby objects. This does not combine objects but applies the state to a group of objects. If one object is visible, nearby objects are also considered visible, and vice versa.
- CellSize (if **MergeInGroups** is enabled): The size of the cells into which the scene is divided. This value should be considered abstract and adjusted experimentally. Increasing this value can proportionally reduce **RaysCount** in **DC_Camera**.

DC_SourceSettings

- ControllerID – The ID of the controller to which the current object belongs.
- SourceType – The type of object (SingleMesh, LODGroup, or Custom).
- CullingMethod (if **SourceType** is SingleMesh or LODGroup): The method used to disable the object. If **FullDisable**, the object is completely disabled; if **KeepShadows**, shadows from the object are retained.
- ConvexCollider - flag specifies whether the created MeshCollider should be convex. This is useful when the source object has a **rigidbody component**. The point is that Unity **does not support non-convex MeshColliders** together with Rigidbody.
- LocalBounds (if **SourceType** is Custom): The bounds of the object. You can set these in the inspector or via special Gizmos in the editor. Note that during the object's preparation stage, bounds will be replaced with colliders.
- OnVisible (if **SourceType** is Custom): Events triggered when the object is visible to the camera.

- OnInvisible (if **SourceType** is Custom): Events triggered when the object is not visible.

DC_Camera

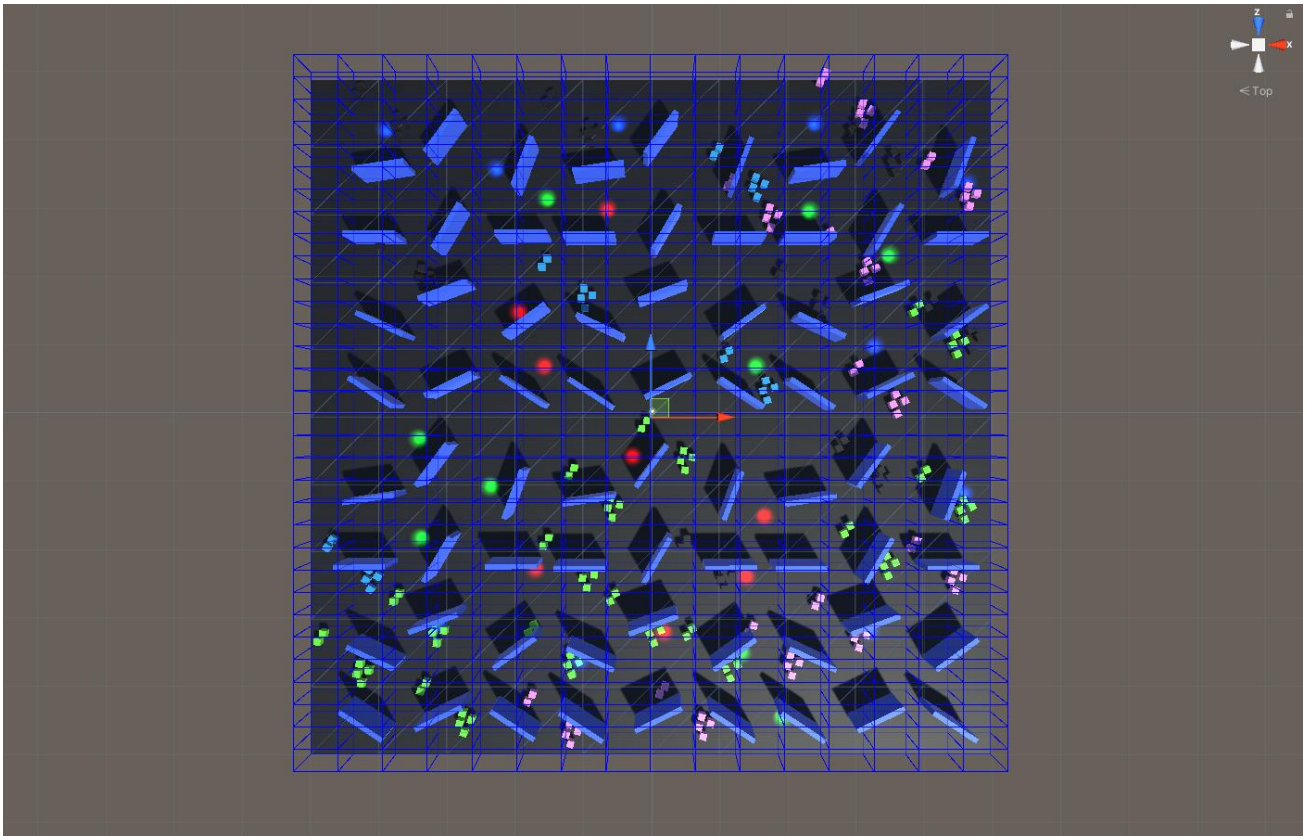
- RaysCount – The number of rays per frame.
- RaysDistribution – The algorithm used to distribute rays across the screen plane.
- AutoCheckChanges – If enabled, **DC_Camera** will additionally check if various camera parameters (e.g., FOV) are changing. This can create additional load. Alternatively, you can manually call the **CameraSettingsChanged()** method when the camera settings change.

Idea (Static Culling)

The idea behind **Static Culling** is to pre-calculate which objects are visible from each part of the scene and then, at runtime, simply toggle objects on or off based on the player's position, without spending time on these calculations.

The solution I implemented involves dividing the scene into cells and calculating the visibility of objects for each cell. At runtime, the current cell of the camera is determined, and the relevant objects are activated.

To avoid artifacts where an object is visible but turned off, objects from neighboring cells can also be considered.



Quick Start (Static Culling)

1) Create a Controller

Create a controller via the menu (Tools -> NGSTools -> AdvancedCullingSystem -> Static).

2) Add Objects

Select the objects you want to cull. In the StaticCullingController, click “Open Selection Tool”. Navigate through the tabs and use the buttons to add objects to the controller.

Note that you cannot add Custom objects through the Selection Tool. To assign custom behavior to an object, attach the **StaticCullingSource** component to the object and set **SourceType** to Custom (see the StaticCullingCustomTargets tutorial in the "Tutorials" folder).

3) Partition the Scene Geometry

Divide the scene geometry so that each cell contains between 10 to 50 objects. Enable **Draw Gizmos** and click **Update** to visualize how the scene is partitioned. It is recommended not to use a **Partition** value greater than 16.

4) Define CameraZones

This section specifies the zones where the camera can be located. Click “Open Selection Tool” in the **Camera Zones** section. In the opened window, click “Create New”. Then select this zone in the hierarchy and use Gizmos to set the boundaries for the CameraZone. You can also set the sizes in the Transform component.

CameraZones should cover the entire area where the camera can be during gameplay.

Specify the **Cell Size**. To see how the scene is partitioned, click **Draw Gizmos** and then **Update**.

Note that rays will be cast from the center of each cell during baking to determine object visibility. Therefore, if you set the **Cell Size** too large, some objects that should

be visible might be turned off. Conversely, if the **Cell Size** is too small, the baking process may take too long.

5) Bake the Scene

If the scene takes too long to bake, click “Cancel” on the Progress Bar and set lower values for **Rays Per Unit** and **Max Rays Per Source**. Then, try baking the scene again.

How It Works (Static Culling)

The Static Culling process is divided into two parts: scene baking in the editor and runtime operation.

In the editor, a **StaticCullingController** is created and configured. Each object participating in culling is assigned a **StaticCullingSource** component, where its behavior is set. **CameraZones** are created, which are bounding boxes defining where the camera can be and storing cells that contain visibility information for objects.

After setting up the scene, the **StaticCullingController** initiates the baking process. The system identifies all objects with a **StaticCullingSource** component to include them in culling and determine how to activate/deactivate them.

The algorithm iterates through all cells in the **CameraZone**, casting rays from the center of each cell to determine object visibility. Instead of casting rays for each object, which would be resource-intensive, the scene is partitioned into a tree structure. The algorithm traverses the tree, calculating visibility for objects in visible leaf nodes and recording data in the **CameraZone** cells.

Once baking is complete, the scene can be run. Each camera rendering the scene and participating in culling should have a **StaticCullingCamera** component. During gameplay, when the camera is within a **CameraZone**, it determines the current cell and activates objects recorded in that cell.

Components Description (Static Culling)

StaticCullingController

- Step2. Partition – The depth of the scene partitioning tree.
- Step3.CellSize – The size of the **CameraZone** cells.
- Step4.RaysPerUnity – The relative number of rays per object. Note that closer objects use fewer rays, while farther objects use more.
- Step4.MaxRaysPerSource – The maximum number of rays per object.

StaticCullingSource

- SourceType – The type of object (MeshRenderer, LODGroup, Light, Custom).
- CullingMethod (if MeshRenderer or LODGroup): Whether to disable the entire object or keep shadows.
- IsOccluder – Disable if the object is transparent or does not block objects behind it (e.g., for windows).
- OnVisible (for SourceType.Custom): Behavior when the object is visible.
- OnInvisible (for SourceType.Custom): Behavior when the object is not visible.

CameraZone

- Set through Gizmos in the Scene window, or you can simply adjust the **Scale** in the Transform component.

StaticCullingCamera

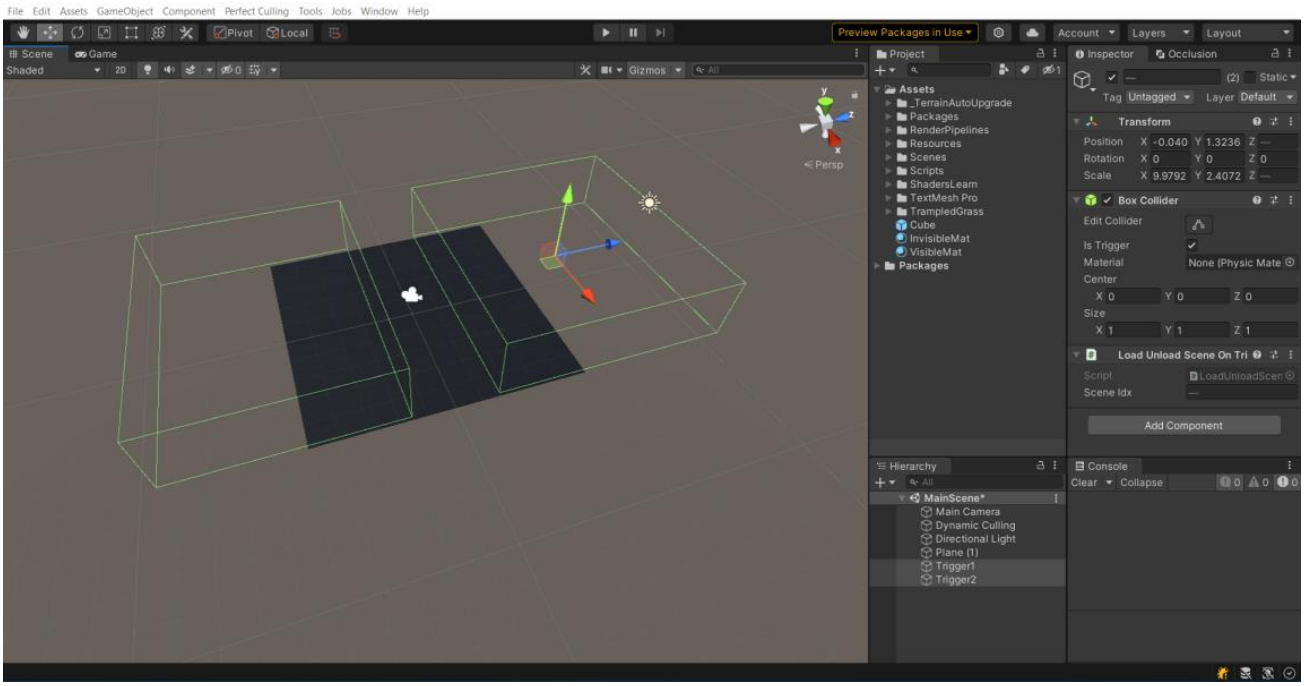
- ShowFrustum – enable to visualise the FrustumCulling process. Objects outside the frustum camera will be disabled. Note that these objects are not rendered by Unity by default, but this option will help you visualise what will be rendered in your scene.
- Draw Cells – visualises the CameraZone cells where the camera is currently located.
- Tolerance – how many cells close to the camera should be included. The larger the value - the more cells, the more objects will be visible.

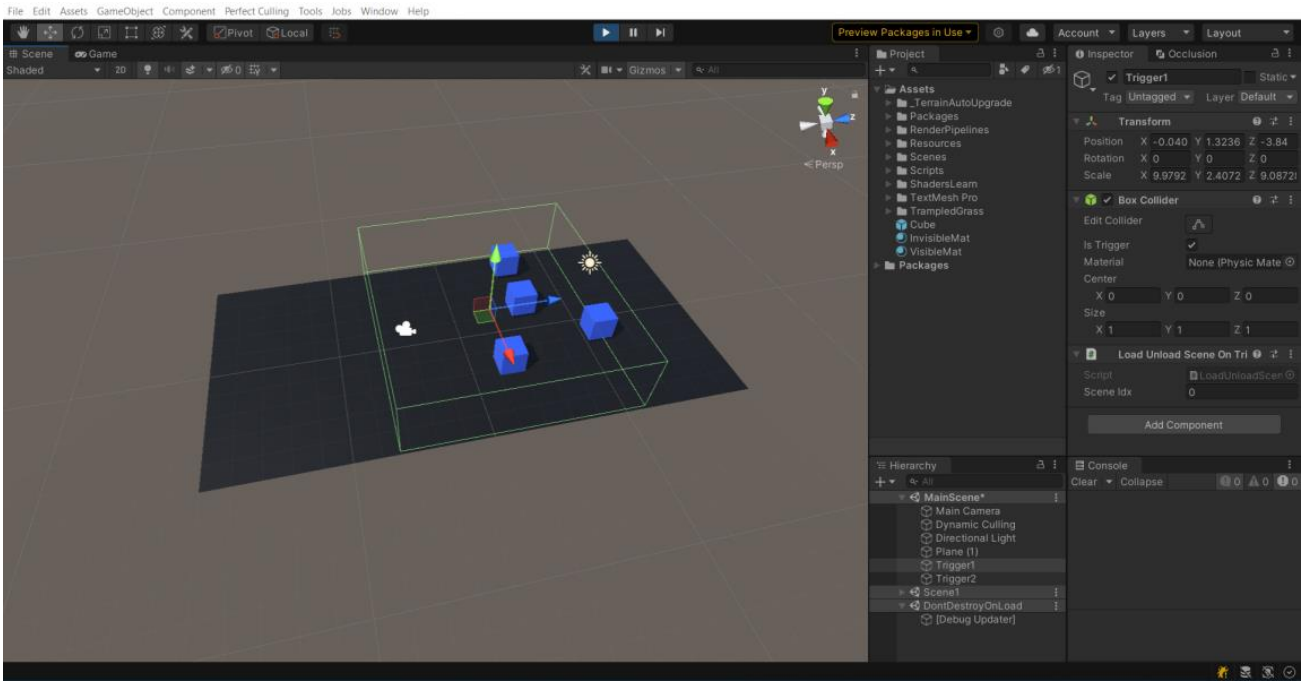
Extra (Additive Scenes)

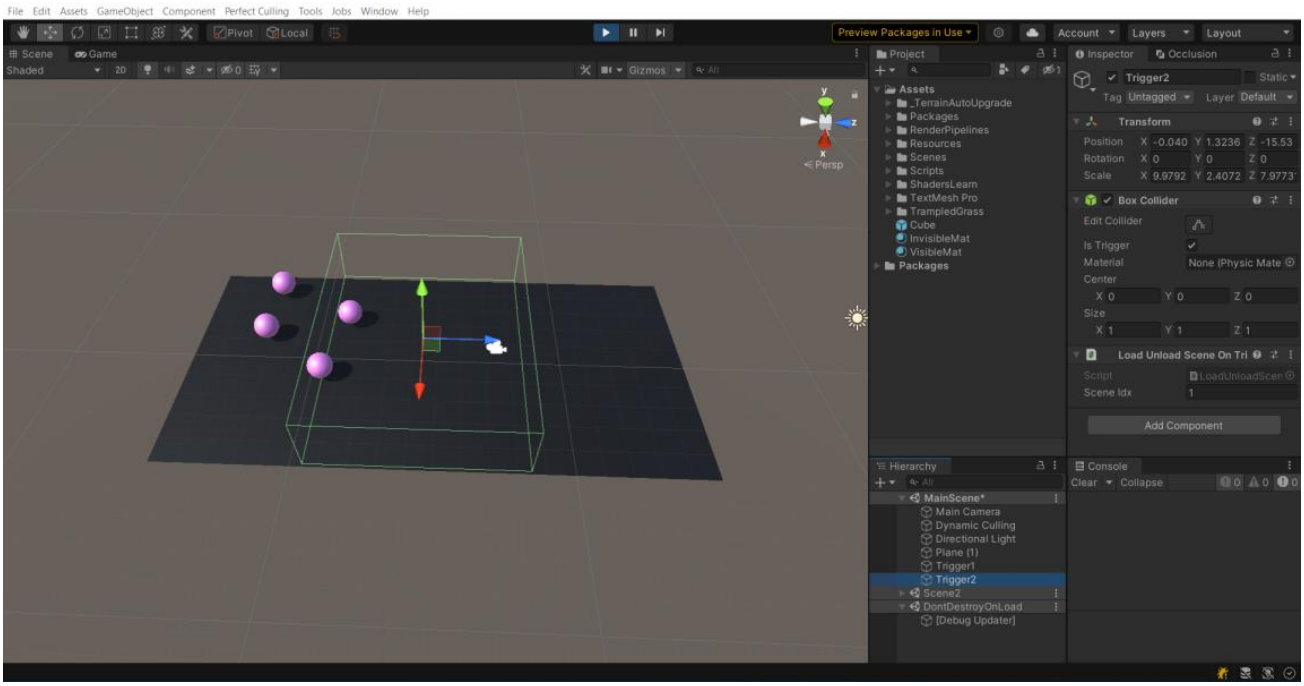
The example is from a discussion in the discord group.

Invite link : https://discord.gg/6EYUn9QhXF

In my example, I have three scenes. Main, Left, and Right. The main scene is loaded first, then if the camera moves to the right then the right scene is loaded, and if the camera moves to the left then the left scene is loaded, if the player leaves the trigger then the scene is unloaded. Scenes can be loaded and unloaded in any order.




File Edit Assets GameObject Component Perfect Culling Tools Jobs Window Help
Pivot Local
Preview Packages in Use
Account Layers Layout
Scene Game
Shaded 2D
Gizmos
Persp
Project
Assets
_TerrainAutoUpgrade
Packages
RenderPipelines
Resources
Scenes
Scripts
ShadersLearn
TextMesh Pro
TrampledGrass
Cube
InvisibleMat
VisibleMat
Packages
Inspector Occlusion
Trigger1
Static
Tag Untagged Layer Default
Transform
Position X -0.040 Y 1.3236 Z -3.84
Rotation X 0 Y 0 Z 0
Scale X 9.9792 Y 2.4072 Z 9.0872
Box Collider
Edit Collider
Is Trigger
Material None (Physic Mate
Center
X 0 Y 0 Z 0
Size
X 1 Y 1 Z 1
Load Unload Scene On Tri
Script LoadUnloadScen
Scene Idx 0
Add Component
Hierarchy
MainScene*
Main Camera
Dynamic Culling
Directional Light
Plane (1)
Trigger1
Trigger2
Scene1
DontDestroyOnLoad
[Debug Updater]
Console
Clear Collapse


File Edit Assets GameObject Component Perfect Culling Tools Jobs Window Help
Pivot Local
Preview Packages in Use
Account Layers Layout
Scene Game
Shaded 2D
Gizmos
Persp
Project
Assets
_TerrainAutoUpgrade
Packages
RenderPipelines
Resources
Scenes
Scripts
ShadersLearn
TextMesh Pro
TrampledGrass
Cube
InvisibleMat
VisibleMat
Packages
Inspector Occlusion
Trigger2
Static
Tag Untagged Layer Default
Transform
Position X -0.040 Y 1.3236 Z -15.53
Rotation X 0 Y 0 Z 0
Scale X 9.9792 Y 2.4072 Z 7.9773
Box Collider
Edit Collider
Is Trigger
Material None (Physic Mate
Center
X 0 Y 0 Z 0
Size
X 1 Y 1 Z 1
Load Unload Scene On Tri
Script LoadUnloadScen
Scene Idx 1
Add Component
Hierarchy
MainScene*
Main Camera
Dynamic Culling
Directional Light
Plane (1)
Trigger1
Trigger2
Scene2
DontDestroyOnLoad
[Debug Updater]
Console
Clear Collapse

The settings I applied to each scene :

1) **Main**. I added there DC_Controller with ID=0, and added camera to the controller.

2) **Left**. I added there DC_Controller with ID=1, added objects from the scene to this controller and made sure that all DC_SourceSettings have ID=1.

3) **Right**. I added a DC_Controller with ID=2, added objects from the scene to this controller and made sure that all DC_SourceSettings ID=2.

Extra (MeshFusion Pro Compatibility)

Although these two tools implement **opposite approaches to optimisation**, sometimes working together can give a **huge performance boost**. There is currently an **alpha version of a script** to make Dynamic Culling and MeshFusion Pro work together.

I plan to put it on the **Asset Store in the future**, but for now you can find the actual version of the script in this **discord channel** named “Useful-Stuff” : https://discord.gg/6EYUn9QhXF

The algorithm of work is as follows :

1) Create MeshFusionController(Tools->NGSTools->MeshFusion Pro)

2) Set MeshFusionSources through the controller or manually, as you like.

3) You need to attach this script to each MeshFusionSource. This can be done quickly. In the "Hierarchy" window, type "t:MeshFusionSource" into the search, select all objects and attach the script to them.

4) Create DynamicCullingController(Tools->NGSTools->AdvancedCullingSystem->Dynamic). You don't need to add objects to it, only cameras.

Now at runtime objects should be combined and culled 😊

Extra (DC_ActivateNearObjects)

Sometimes it is useful to enable all objects within a certain radius of the camera, even if they are overlapped by other objects.

This is a small script that can be found in the folder : AdvancedCullingSystem -> Core -> Runtime -> Utils -> DC_ActivateNearObjects

Just attach this script to the camera