

Table of contents

Introduction	1
Optimizing Unity games	2
Rendering performance	2
Script performance	3
Physics performance	3
What is this all about?	4
How does M2HCullingManual work?	4
How does M2HCullingAuto work?	6
How well does the culling system work?	7
Culling system license & final words	9
Appendix A: Project scenes overview	10
Appendix B: A thorough look at the code	11

Introduction

Having finally realized your ambitious plans for a game, you might face big performance problems; Now that everything works it's running terribly slow at the clients computer. This document, and accompanying Unity project, helps you address performance problems in your Unity application. I'll start with a brief introduction to general optimizing techniques for Unity games after which I will show you my Unity occlusion system which greatly improves rendering performance. You can this occlusion system in your own (commercial) projects. Haven't got the accompanying unity project? Download it here: <http://www.M2H.nl/unity>

Enjoy,
Mike Hergaarden
 21 June 2010

Optimizing Unity games

To optimize the speed of your Unity application the following parameters matter:

1. Rendering performance
2. Script performance
3. Physics performance

For all of these items I will give you some quick pointers so that you know how to optimize your game; whatever is slowing it down.

Rendering performance

Unity manual "Optimizing graphics performance"

Unity's own optimization tips, a must read:

<http://unity3d.com/support/documentation/Manual/Optimizing%20Graphics%20Performance.html>

Combine meshes

Combines meshdata of several meshes that use the same material to greatly reduce drawcalls. Sometimes it's wise to even consider combining textures so that you can make use of combine meshes more effectively. See the Unity Standard Assets for the script.

Render stat window:

While the built in render stats window does not exactly show you your performance bottleneck, it can help you pinpoint it.

See: <http://unity3d.com/support/documentation/Manual/RenderingStatistics.html>

Quality settings

Unity provides you with 6 standard quality settings levels that change some basic, yet very important, quality settings right away. You can create a runtime script to dynamically change this setting depending on the users framerate.

See: <http://unity3d.com/support/documentation/Components/class-QualitySettings.html>

Guideline for Character models:

See: <http://unity3d.com/support/documentation/Manual/Modeling%20Optimized%20Characters.html>

Camera layer culldistance:

A handy function to define culling distance per layer (instead of using just the camera far plane setting). You wouldn't need this if you use my culling system though.

See: <http://unity3d.com/support/documentation/ScriptReference/Camera-layerCullDistances.html>

Frustum culling

Per default Unity applies frustum culling for you; it removes all objects that are outside of the camera's view frustum. However, objects that are inside the view frustum but behind other objects (and thus not visible) are still being rendered and cost you performance. This is where my occlusion system can help you boost your FPS.

Script performance

Script optimization

A must read: http://unity3d.com/support/documentation/ScriptReference/index.Performance_Optimization.html

I have only a few additions to Unity's script optimization guidelines:

1. Do not abuse OnGUI or FixedUpdate for non-GUI functions.

Don't use these functions for stuff that could be run in Update instead. Abusing these functions will really cost performance because OnGUI and FixedUpdate will start hogging the application; if they run too often they take up Update()'s time.

2. Unity profiler (Pro only)

Use the profiler to detect bottlenecks.

See <http://unity3d.com/support/documentation/Manual/Profiler.html>

Physics performance

Rigidbody sleeping:

Disable physic calculations below a certain velocity.

See: <http://unity3d.com/support/documentation/Components/RigidbodySleeping.html>

Solver iteration count:

How accurate do you need your physics calculations? The more precise, the heavier the calculations. See: <http://unity3d.com/support/documentation/ScriptReference/Physics-solverIterationCount.html>

M2H Occlusion culling system

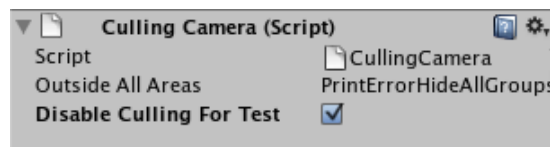
What is this all about?

The M2H Occlusion culling system disables rendering of object that are not visible and therefore greatly reduces draw calls and the verts/tris count. However, the real power behind this tool is the Editorscript for the authoring side. I have made two versions of this system, let's call them M2HCullingManual and M2HCullingAuto. The difference with Auto is that it requires less setup; the system calculates which objects are in the areas. If you need more control you can use manual. I'll start explaining the manual system.

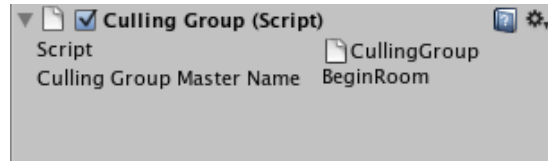
How does M2HCullingManual work?

The culling system uses three concepts; the camera culling script, culling areas and culling groups. For every culling area you select which culling groups are visible and, optionally, which are never visible. The culling groups are enabled/disabled(culled) depending on the camera's position. The actual culling is done by enabling/disabling unity render components.

Let's go over all system settings in a situation where we've defined some culling groups and areas (in this case we're using the scene "SimpleExample_Manual").

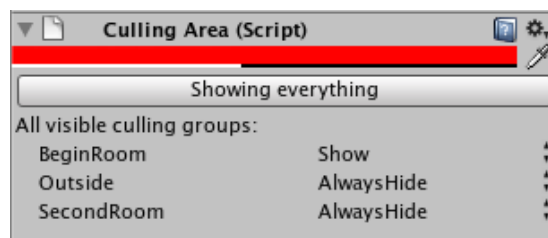


Firstly, the camera has the CullingCamera script attached. This script processes all culling and disables renderers of object that aren't visible. We've got two debugging options:
Outside all areas: specify what should happen when the camera is not inside any culling area. It's best practise to leave this option to "Print error and hide all groups" so that it's very clear when you're not covering your entire level. If you somehow don't need culling in your entire world you can change this to just "Hide all".
Disable Culling for test: The second option can be used for debugging to check how much performance the culling saves you. Don't forget to disable this option after testing!



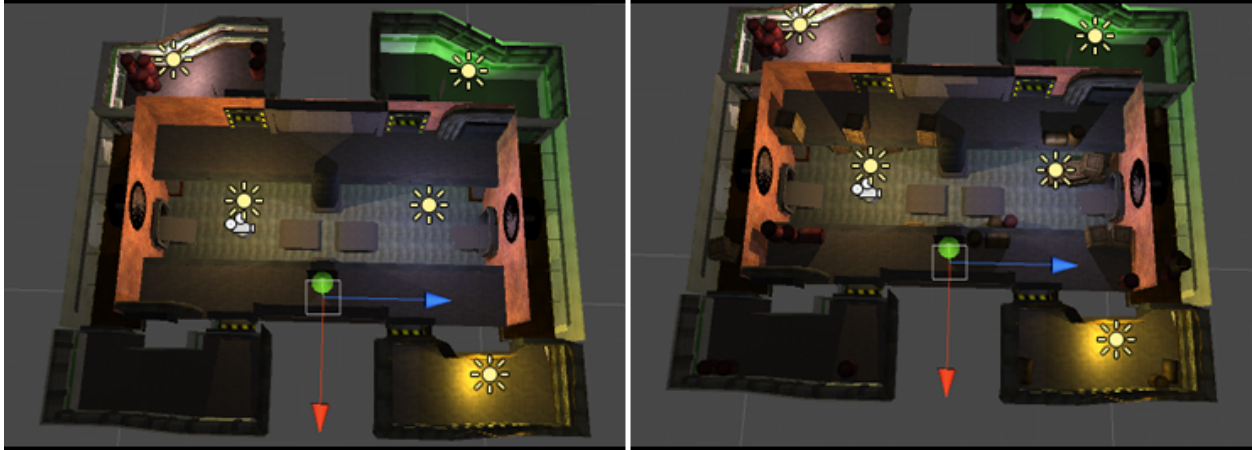
The culling group script is attached to the parent of all game objects you want to contain in this group. This culling group can be given a custom name. Per default it will copy the gameobject name to save you work.

Tip: You can give several groups the same master name; they will all share the same culling options!



This is the most interesting script. This script is attached to empty game objects to define areas for your camera with culling settings.

- You can customize the gizmo color however you like.
- The button "Showing everything" can be clicked to show only the groups that are visible by this area's settings. The button will then switch to "Showing only this area". The button's option will be remembered for all Culling Areas so that you can quickly check out all areas individually.
- Per culling group, select the desired option for this area. Note that you can leave options blank as groups are hidden by default. For clarity (and to save you work) you should also leave options blank when other (overlapping) areas have already defined the right action for a certain group.
- **Tip:** Clicking on a Culling group will hide all culling groups except the one you selected to quickly show you the culling group.



Occlusion culling in the factory example scene. The factory mesh is unoptimized, it doesn't allow for culling and has too much materials. We only use culling on the level objects here.

Every frame the culling is calculated in two steps:

1. All objects are visible as usual, *unless* they are children of a gameobject with the CullingGroup.cs script on it; those are hidden by default. The occlusion culling is only applied on these gameobjects.
2. The camera checks in which CullingAreas it is enclosed. The settings of each of these areas that encompass the camera are used; The camera takes a union of all visible groups, all visible groups are enabled. Do note that if at least one area has a group marked as 'Never show', it will not be shown.

How does M2HCullingAuto work?

One possible improvement of the Manual occlusion system is having the system calculate the objects inside areas automatically by using the `renderer.bounds` that Unity provides us. This is quite easy, fast and saves you the hassle of adding and sorting CullingGroups. However, the downside is that you need to decide what transforms you take into account. If you'd assign a dynamic transform (enemy, player) to a certain area, it would mess up once it moves to another area. For this system need to know the difference between static and dynamic objects, after which we can calculate the areas of all static objects. I have implemented this by requiring you to add all static objects as child of a gameobject called "CullableObjects".

The automatic culling system works exactly like the manual system. In fact; I used the

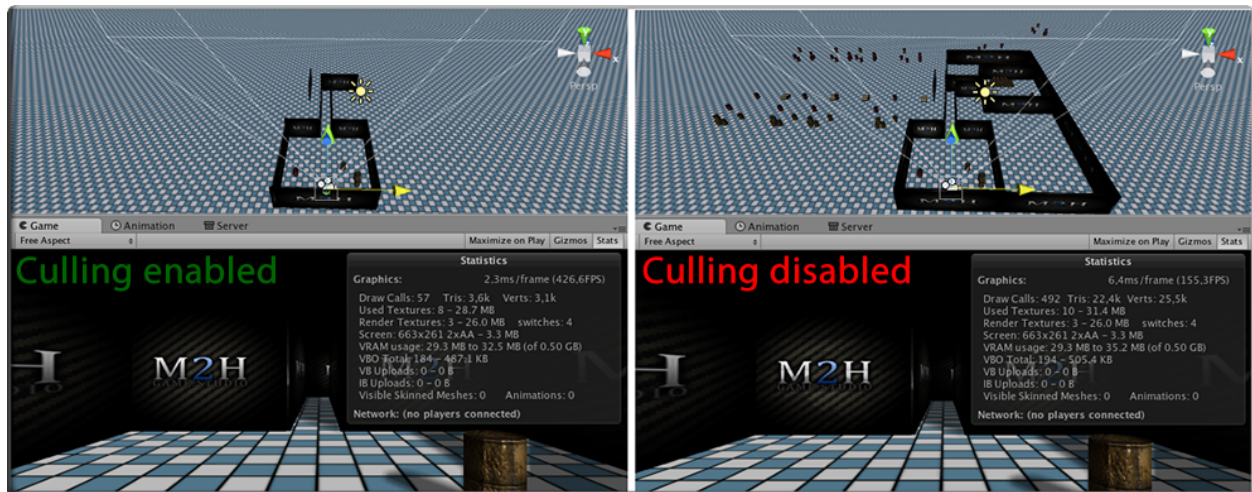
manual system as a base for the automatic system. The only change is the removal of the CullingGroup script to save you some authoring; CullingAreas now automatically calculate which objects are inside their bounding volume. You now define which areas are visible from a certain area. By default all objects inside an area are visible in that area.

Every frame the culling is calculated in two steps:

1. All objects are visible as usual, *unless* they are children of the gameobject "CullingObjects" those are hidden by default, we only apply culling over these objects.
2. The camera checks in which CullingAreas it is enclosed. The settings of each of these areas that encompass the camera are used; The camera takes a union of all settings. Do note that if at least one area has an area marked as 'Never show', its objects will not be shown.

How well does the culling system work?

How effective the culling works really depends on how you set it up and how well you've designed the level to allow culling. The more you want to cull, the more effort it takes to set it up. You need to find a balance. As shown below, proper culling can make a huge difference. In this example we could use combine children to decrease the drawcalls even more. Note that there is no performance difference in using the manual or the automatic system; they differ only at the authoring side.



Culling differences: Before: 155FPS, 492 drawcalls 22,4k Tris 25.5k Verts
After: **426FPS**, 57 drawcalls 3,6k Tris 3,1k Verts

How do I use the system in my projects?

You can choose between the manual system and the automatic system; I advice the automatic system as it requires less setup time.

Manual system:

Copy the "M2HCullingSystem_Manual" folder to your project.

1. Attach the CullingCamera_Manual.cs script to the camera that you want to use culling on.
2. Define groups of object that can be culled (e.g. per room/hallway in your game), make all of them children of an empty game object. Name it clearly and add the CullingGroup_Manual.cs script to this parent.
3. Create new gameobjects and add the CullingArea_Manual.cs script to them. For each of the areas setup what they can see or can never see.

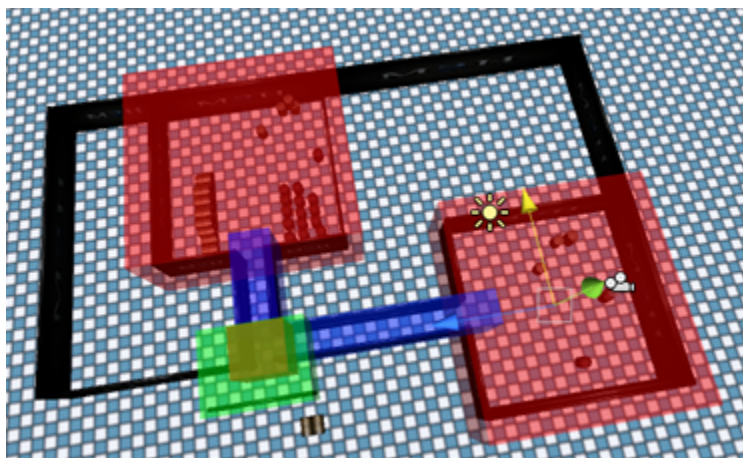
Automatic system:

Copy the "M2HCullingSystem_Auto" folder to your project.

1. Attach the CullingCamera_Auto.cs script to the camera that you want to use culling on.
2. Add all cullable object as child of a gameobject called "*CullingObjects*"
3. Create new gameobjects and add the CullingArea_Auto.cs script to them. For each of the areas setup what they can see or can never see. Make sure that these cullingareas cover all cullable objects.

Both systems:

Make smart usage of Culling Area overlapping to save you setting the same options repeatedly. Also don't feel obliged to size the areas precisely; it's better to have too big areas than too small areas.



Overlapping culling areas

Culling system license & final words

You're free to use and adapt this system however you like in any project. However, there are just three restrictions.

If you use it for a commercial project I ask you to donate an amount between 5 - 100 Euros to support@M2H.nl (Paypal). You decide what it's worth to you. After this donation you're free to use this system in any of your commercial projects. Lastly you are not allowed to resell this script or remove the license notes at the top of the sourcefiles.

I hope this document helped a great deal. If this document/system was of great help to you it would be nice to hear!

For more Unity tutorials/examples see: www.M2H.nl/unity

Appendix A: Project scenes overview



Examples/FactoryExample/Factory_Automatic

This scene's hierarchy consists of four main gameobjects:

CullingAreas

The children of this GameObject define all CullingAreas and their settings. These areas have the CullingArea_Auto script attached.

CullingObjects

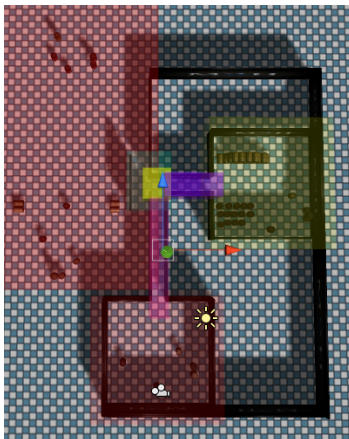
This GameObject contains all objects that should use culling.

First Person Controller

Unity's default FPC with some adjustments. E.g. the CullingCamera script has been added to the camera.

Level

Contains the level mesh and lights. Note that this level mesh is not optimized for culling as it's one big mesh.



Examples/FactoryExample/Factory_Manual

This scene is exactly the same as the Factory_Automatic scene, except for a different culling setup; The CullingArea_Auto and CullingCamera_Auto scripts have been replaced with their manual-culling equivalent. The CullingObjects has been renamed CullingGroups and all objects are now sorted per group, the parents of those groups have the CullingGroup_Manual script attached.

Examples/SimpleExampleScene/SimpleExample_Auto

Examples/SimpleExampleScene/SimpleExamples_Manual

These two scenes use the exact same hierarchy as the Factory scenes. Obviously only the level and its culling areas are different. On the image to the left you see a visualisation of the CullingAreas.

Appendix B: A thorough look at the code

This appendix features an extensive look at the inner workings of the automatic culling system. The most important code lines are shown here, with additional comments. As the entire culling system is all about the camera, the culling system structure is best explained by starting to look at a simplified version of the CameraCulling script. The CameraCulling takes a union of all CullingArea settings of the area's that contain the camera. For performance reasons the culling is recalculated only after moving more than a certain threshold.

CullingCamera_Auto.cs

```
public enum CullingOptions_Auto { _, Show, AlwaysHide, } // _ is nothing (hide)

void OnPreCull() {
    //Check if we need to recalculate the culling. If not, return.
    ....

    //Set settings to default
    foreach (CullingAreaSettings_Auto aGroup in currentCameraSettingsForAllAreas){
        aGroup.cullingOptions = CullingOptions_Auto._;
    }

    //Now, check what areas we can enable: feed the master list
    .....
        if (bounds.Contains(relative))
        {
            //Enable all areas that you're allowed to see here
            thisAreaList.Add(area);
        }
    .....

    //Retrieve settings of all areas we're inside of.
    foreach (CullingArea_Auto area in thisAreaList){
        weAreInAtLeastOneArea = true;
        EnableGroupsFromCollider(area);
    }
}
```

```

//Enable all enabled areas
    foreach (CullingAreaSettings_Auto liveCullGroup in currentCameraSettingsForAllAreas){
        if (liveCullGroup.cullingOptions == CullingOptions_Auto.Show || (!weAreInAtLeastOneArea &&
(outsideAllAreas == CullingCameraOption_OutsideAllAreas_Auto.PrintErrorShowAllGroups || outsideAllAreas ==
CullingCameraOption_OutsideAllAreas_Auto.ShowAllGroups)))
            {
                StartCoroutine(liveCullGroup.script.StopCulling());
            }
    }

//Disable areas that should never be shown (override)
    foreach (CullingAreaSettings_Auto liveCullGroup in currentCameraSettingsForAllAreas)
    {
        if (liveCullGroup.cullingOptions == CullingOptions_Auto.AlwaysHide)
        {
            StartCoroutine(liveCullGroup.script.StartCulling());
        }
    }
}

void EnableGroupsFromCollider(CullingArea_Auto area)
{
    //Copy this areas settings to the masterlist (currentCameraSettingsForAllAreas).
}

```

Next are the CullingAreas. Remember that a CullingAreas is nothing more than a box with rules that define what areas should be visible from inside this area. Furthermore, a CullingArea calculates which objects are inside it.

```

CullingArea_Auto.cs
public void SetupVars(bool forceRecalculate){
    //Check what objects are inside this area and cache their Renderer components
}

public IEnumerator StartCulling()
{

```

```

    //Disable all this areas renderers
    ....
    foreach (Renderer myRenderer in myRenderers)
    {
        myRenderer.enabled = false;
    }
}

public IEnumerator StopCulling()
{
    //Enable all this areas renderers
    ...
    foreach (Renderer myRenderer in myRenderers)
    {
        myRenderer.enabled = true;
    }
}

```

The code above covers entire culling system already. However for easy authoring there's an editor script to override the CullingArea editor inspector view to define the settings per CullingArea. This editorscript also allows you to preview just one CullingAreas objects or all enabled CullingAreas for the CullingArea you are currently editing.

CullingAreaEditor_Auto.cs

```

void Awake()
{
    //Have Unity notify us if we're switching play mode
    EditorApplication.playmodeStateChanged = SwitchingToPlay;
}

void OnEnable()
{
    //Load all settings: Make a list of all other areas and set default to hide. Sorted alphabetically.
}

void OnDisable()
{

```

```
//In case we were previewing the culling, show all hidden objects again.
}

void SwitchingToPlay(){
//In case we were previewing the culling, show all hidden objects again.
}

void ReloadAllAreas(){
//Feed the list of other areas and it's own settings
}

public override void OnInspectorGUI(){
//Visualise the settings for all other areas
}

void EnableAllObjects(){
//Calls StopCulling on all CullingAreas
}

void ShowOnlyGroups(CullingAreaSettings_Auto showOnlyGroup){
//Culls ALL areas except the one passed as parameter (for previewing)
}

void ShowOnlyThisArea(){
//Culls all areas except this one
}

void ChangedSetting(CullingAreaSettings_Auto maingroup){
//If a preview is on; apply the new changes.
}

void AddIfNotExists(CullingArea_Auto anAreaScript, List<CullingAreaSettings_Auto> oldList)
{
// Adds the CullingArea to the currently editing CullingArea
// Copies previous settings if they exist (oldList).
}
}
```