MANUAL

Table of Contents

Introduction
Overview
Setting Up
Occluders
Shadows and Occlusion
LODs
    Creating LODs
    LOD Selection
Optimization
    Basics
    Controlling the Hierarchy
    MultiThreading
Multiple Active Culling Cameras
Umbra Comparison

Questions or problems?

support@makecodenow.com

Introduction

SECTR VIS is a high performance, low memory, occlusion culling solution for Unity, with support for the complete set of Unity rendering primitives including lights, shadows, particles, meshes, and terrain. SECTR VIS is fully dynamic, requires no tedious baking process, and works with both Unity Free and Pro. Heavily optimized, SECTR VIS is a great solution for all platforms, especially tablets, smart phones, portables, and "last gen" consoles.

SECTR VIS is designed to make the process of adding occlusion culling simple, straightforward, and completely under your control. Initial setup of a scene takes just minutes, after which you can easily refine the culling to produce the mix of memory, CPU, and GPU that's right for your game.

SECTR VIS works with any Unity scene, but is designed primarily for indoor or hybrid-indoor (think most popular first and third person games) games. Only truly outdoor, fully open world games are not a good fit for SECTR VIS.

All SECTR modules include complete source code, online support, and are fully compatible with Unity, Unity Pro, and Unity Version Control.

Overview

SECTR VIS is a highly optimized implementation of recursive, portal based occlusion culling. The culling starts when a SECTR CullingCamera component is added to one of the Cameras in a scene w/ Sectors and Portals. Before the Camera renders, the CullingCamera does its work.

The following is a basic outline of the culling algorithm:

1. Starting in the current sector, the CullingCamera tests its frustum against all of the renderable objects in that Sector, hiding any that are not visible.
2. Next, the CullingCamera tests its frustum against all of the Portals leading out of that room. If any portals are visible, a new frustum is generated by performing a geometric intersection of the camera frustum and that Portal, dramatically winnowing down the size of the frustum.
3. This new, clipped frustum is then passed into the Sector on the other side of the portal, and the objects in that Sector are tested as in Step 1, and the Portals in Step 2.
4. This process continues until there are no visible portals. Usually, this process terminates quickly due to the natural layout of game spaces.

Because all of this logic is implemented during PreCull (and cleaned up in PostRender), it's completely safe to have multiple CullingCameras active in a scene at once (i.e. for split-screen multiplayer or PiP gameplay).

SECTR VIS also supports Occluders, which can be thought of as anti-Portals. Where a Portal allows visibility to pass, Occluders prevent visibility from passing, hiding any objects behind them. Occluders are discussed in greater detail in their own section below.

The entire SECTR VIS module is designed for maximum simplicity and controllability, with as little "automagic" as possible. This means that it takes more to enable culling than the simple press of a button, but it also means that you can control the balance memory, CPU, and GPU that is right for your game, on the platforms you care about.

Setting Up

Preparing a scene for SECTR VIS culling is easy. First, you need to have a scene with some Sectors and Portals in it. For new users, please see the SECTR CORE documentation for more detail on how to do that.

Once your scene has Sectors and Portals in it, enabling culling is as easy as adding a CullingCamera component to the main camera of your scene. If you have multiple game cameras, each one can have a CullingCamera component. They will not interfere with one another.

If you want to see the effects of the culling in the Unity Editor, select the GameObject with the CullingCamera component on it and make sure that the Gizmos are drawing for CullingCamera. As you move the camera around the scene, you'll see the debug rendering of the frustums as they propagate through the Portals. If you run the game in the editor and then select the CullingCamera, you'll actually see objects appear and disappear as they are culled.

One common question when setting up a scene for occlusion culling is which objects should be children of Sectors and which should have their own SECTR Members. SECTR VIS was designed to make this process as simple as possible, provided you follow two simple rules:

- Objects that move between sectors should have a Sector Member component.
- Objects that extend beyond bounds of their Sector should have a Sector Member component.

Following these two rules will ensure that the visibility calculations are always correct, even in tricky situations like objects outside a Sector casting shadows into it.

Occluders

While Portals do most of the culling work, there are cases where they are not completely effective on their own. In particular, large, open spaces (like huge rooms our outdoor sections of hybrid-indoor games) may have a large number of objects, but few natural windows or doorways. Fortunately, these scenes often have large, solid objects in them, and that's where Occluders come in.

Where Portals allow visibility to pass through them, Occluders block visibility, and hide all of the objects behind them. This makes them a good fit for large, solid objects in big spaces. These large objects hide many objects and can be used for culling. To do so, simply add a SECTR Occluder component to the object in question and draw a mesh for it.

Like Portals, Occluders in SECTR VIS are required to be convex and planar (for performance reasons). This requirement works well for some occluders (like a big wall) but may not work well for others (like a cylindrical column). To handles cases like columns, Occluders can be set to auto-orient to the culling camera, and users can specify which axes should auto-orient. This makes it possible to efficiently represent a much larger range of occluded shapes. For example, to represent a column, make a rectangular mesh for the Occluder and set it to auto-orient about its Y axis.

Shadows and Occlusion

SECTR VIS is designed to accurately and efficiently properly cull all shadow casting lights. The one tricky case for SECTR VIS is handling shadows cast by directional lights, because they cast very long shadows and have infinite bounds. SECTR VIS supports shadow casting directional lights, but in order to keep performance high, requires one additional piece of human intervention.

Each Sector and Member has an attribute for a Directional Shadow Caster. If the object that this Member is part should cast shadows from a directional light, simply assign that object to the Directional Shadow Caster attribute. This works because Unity will only cast shadows from a single directional light at a time anyway.

You may also wish to tweak the Directional Shadow Distance attribute in the Shadow Culler. This value determines how much to the object's render bounds will be extended during culling. The value must be high enough that it matches the maximum distance of the cast shadow. However, excessively large values will waste CPU time.

You can change the Direction Shadow Caster at any time, just make sure you use the light that's actually casting the directional shadows.

LODs

SECTR VIS includes support for dynamic LOD (level of detail). LODs are most commonly used to replace high poly geometry with lower poly substitutes when an object is far from the camera, but SECTR VIS LODs can be used to control any game object, including ones with lights, particles, collision, etc.

Creating LODs

Creating LODs is as simple as adding a SECTR LOD component to any game object. Once added, use the Inspector to add LODs and specify which children of the node should be included in each LOD. Children will be activated when their LOD is in range, and deactivated when it is out of range. Any child that is not part of any LOD will be unaffected by the LOD system.

Once your LOD components are created, make sure to add a SECTR Culling Camera component to the scene camera. The Culling Camera is responsible for actually updating LODs every frame. Note that SECTR LOD components are compatible with the SECTR occlusion culling but do not require Sectors or Portals to work.

LOD Selection

The current LOD is selected based on the screen space size of the object. LOD thresholds are expressed as a percentage of screen space size. For example, a threshold of 50% for LOD0 means that LOD0 will be active whenever the object is 50% of the screen or larger.

The LOD bounds are computed based on the union of all renderable objects in all of the LODs. The LOD bounds are separate from the Member bounds in order to ensure that they are stable and conservative, both of which are required for high quality LODs.

Optimization

SECTR VIS tries to be as efficient as possible "out of the box" but there are some additional steps that users can take to squeeze more performance out of the system.

Basics

If any Sector, Portal, or Occluder will not move during gameplay, you can mark it as Static and get some additional CPU savings. Just make sure that this object really will be Static.

SECTR VIS works with Portals with any number of sides. However, the more sides the Portal has the more expensive it is to cull. Portal geometry does not need to exactly match the visible geometry, so you can save CPU time by simplifying the silhouette of your Portal.

Controlling the Hierarchy

SECTR VIS will cull any object that has a Sector or a Member component on it, as well as all of the children of that object. Sector and Member have an attribute called Child Culling that determines how that group of objects will be called. The value you assign to Child Culling determines if each child will be culled based on its individual bounding box or if children will be culled based on their aggregate bounds. Individual culling is more accurate, but requires more CPU. In some cases, the marginal benefit of more accurate culling is not worth the extra CPU cost of the detailed culling. The default behavior is to cull individual children for Sectors and to cull the group as a whole for Member. You can override these behaviors and even add new Member components to control the granularity of the culling.

One example of this might be on lower power platforms (like smartphones and tablets). On these platforms, it may be desirable to mark the Cullers of each Sector to not cull individually. That means Sectors will only be culled if they are completely invisible, but that kind of gross rejection may be worth the savings of detailed culling.

Another example is culling a pile of objects, like a stack of boxes. Even on high end machines, it may not be worth the CPU time to cull each box individually. Instead, it would be sufficient if they render or cull together. To implement this, simply create a new empty transform for the pile of objects, add a Culler component to that new object, and set Cull Individually to false for the new Culler.

MultiThreading

SECTR VIS can use multiple threads to perform occlusion culling in parallel. On games with complex scenes and multiple CPUs, this can yield a significant performance increase. To enable multi-threaded culling, simple set the Num Worker Threads variable to a value greater than 0 in the Culling Camera component.

Some care should be taken when enabling multi-threaded culling. The Unity/Mono scheduler is only one of many systems that use threads, and it is possible for threads created by SECTR to be halted while other critical work happens. If a SECTR thread is halted for too long, it can cause a spike in the amount of time spent culling which will feel like a hitch to the player. Usually it's best to start testing with 1 or 2 threads, and to pick the number of threads based on the number of player CPUs for production games.

Multiple Active Culling Cameras

If you want to have multiple cameras that need independent occlusion culling (like a picture-in-picture security cam), you have a few options, each of which work around an issue in Unity's multi-camera culling:

- Go to SECTR/Code/Vis/Scripts/SECTR_CullingCamera.cs and uncomment LAYER_CULL_RENDERERS and LAYER_CULL_TERRAIN.
- Create a new project layer called InvisibleCulling.
- Select your culling cameras and set the Invisible Layer property to the InvisibleCulling layer.
- If you have lights and renderers on the same GameObject, make sure that the bounds of the light is the same size or smaller than the renderer. If you don't, you may see visual glitches when one of them culls unexpectedly.

Note that for games that use multiple cameras for UI, FPS weapons, etc. you can ignore this section as long as only your main camera needs scene culling.

Umbra Comparison

No discussion of SECTR VIS would be complete without an objective comparison with Umbra, the built in Occlusion system for Unity Pro. The following are some of the major differences between the products:

- The first, and for some people most important, difference is that Umbra requires Unity Pro, while SECTR VIS works with Free or Pro.
- SECTR VIS is designed for indoor and hybrid-outdoor games, where Umbra works for fully outdoor and fully indoor games. Umbra is especially effective where SECTR VIS is weakest – fully outdoor environments of the kind you see in open world games.
- Umbra requires a baking step. While this has gotten much better since 4.3, it can still take time on large scenes. SECTR VIS requires no baking at all.
- Umbra has some dynamic elements (if you use their helper components), but they are not well explained and they have dependencies on some of the static, baked data. SECTR VIS is completely dynamic, so if you can create it, you can change it during the game.
- Umbra does not work with scene streaming (as implemented in SECTR Stream). SECTR VIS does, for obvious reasons.
- Umbra is implemented in C/C++ and has access to native Unity code. SECTR VIS is implemented in C#, and does not have super low level access to engine or hardware. While Umbra and SECTR VIS perform comparably in many test cases, Umbra's baking step and low level access mean it will likely always have the raw speed advantage.
- Umbra is closed source to Unity developers, while SECTR VIS includes all source as part of its license.
- Umbra is designed to be an auto-magic solution. It requires nothing more than a button press to get working, but it can be difficult to control if that auto-magic solution does not produce the results you want. SECTR VIS requires some human work to setup, but once set up does exactly what you tell it to. The products make different tradeoffs here, and which is right for you, only you can say.