CORE

MANUAL

Table of Contents

Questions or problems?

support@makecodenow.com

Introduction

SECTR CORE is the foundation of the SECTR suite of Unity extensions. It includes all of the tools necessary to quickly and easily add Sectors and Portals to your Unity-based game, as well as full source code that you can build your own unique features.

SECTR CORE is offered to the Unity community free of charge. If you like the quality of the tools and code in SECTR CORE, we encourage you to check out our paid modules:

- **SECTR Audio:** Brings the latest, cutting edge audio production tools and technologies to Unity, including an unparalleled suite of editor extensions and runtime components that let you create rich, complex soundscapes with ease and play them back with a minimum of CPU overhead.
- **SECTR Stream**: Makes it easy to save memory, increase performance, and decrease load times by splitting your scene into multiple chunks and streaming them in realtime.
- **SECTR Vis:** A high performance, low memory, fully dynamic occlusion culling solution for Unity, with support for the complete set of Unity rendering primitives including lights, shadows, particles, meshes, and terrain.
- **SECTR Complete:** All of our current (and future!) SECTR modules and some unique components besides. Cheaper than buying each module separately.

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

Overview

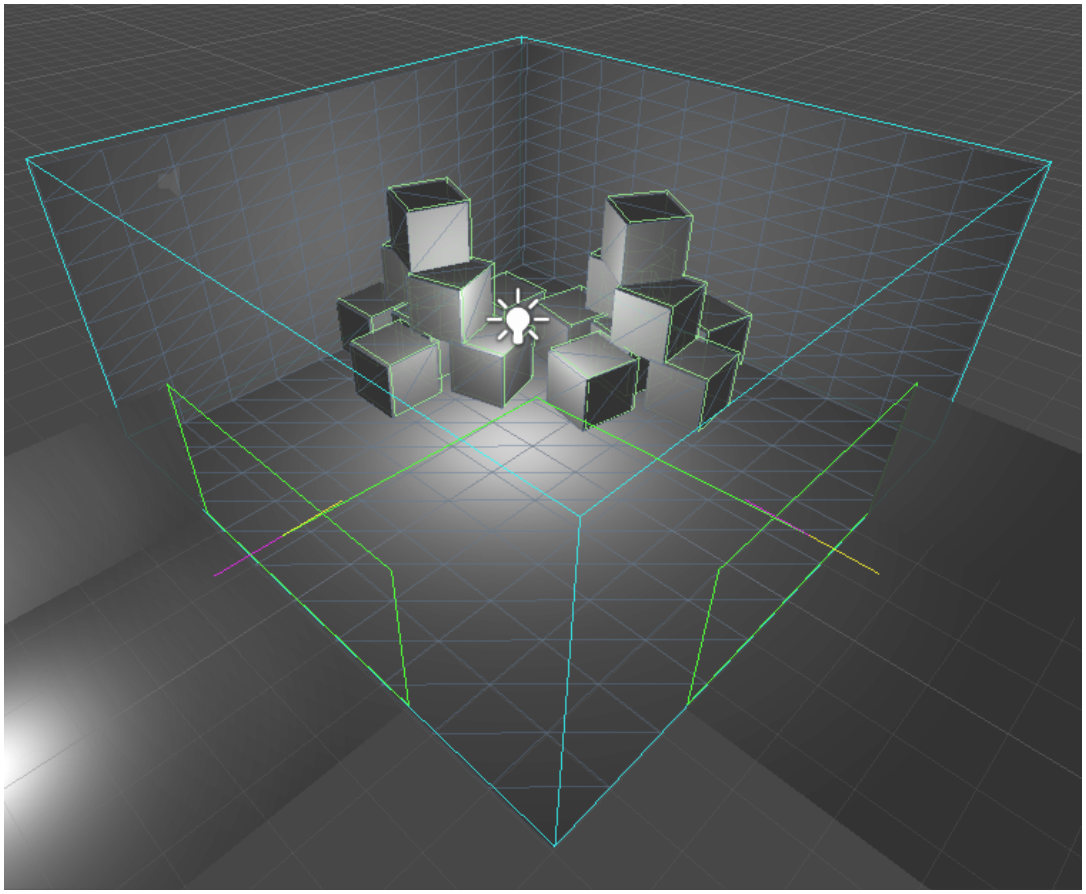
Game developers know that most levels can be broken down into spaces (Sectors) and the connections between them (Portals). This pattern works for indoor games, but also hybrid-indoor games, where exteriors are really just large rooms (i.e. Dead Space, Metroid Prime, Gears of War, etc). Over the years, AAA developers have learned how to use this structure to create games that run faster, sound more realistic, and look better than they otherwise could. Unfortunately, Unity doesn't have any built in tools for creating these kinds of logical spaces, which is why we created SECTR.

As the foundation of the SECTR framework, SECTR CORE provides all of the tools and code necessary to add Sectors and Portals to Unity scenes. SECTR CORE also includes a suite of libraries for interacting with and extending this framework, including graph traversal and geometric operations.

The building blocks of SECTR CORE are Sectors (spaces), Portals (connections), and Members (objects in Sectors). When Sectors are connected through Portals, they form a graph, called the Sector/Portal Graph. Each of these key components are discussed in their own section below.

Sectors

A Sector represents a volume of space, and the objects within that space. In most games, Sectors will be rooms and hallways, but Sectors can represent anything from a section of an outdoor level to a bonus area in a side scroller. Sectors are connected by Portals (described below).



Static and Dynamic

Sectors can be marked as Static or Dynamic, but default to being Static (since most games do not have moving rooms). When marked as Static, the Sector will save some CPU time by not computing its bounds every frame. Generally speaking, any children of a Static Sector either be Static themselves, have a Member component on, or otherwise be guaranteed to stay within the bounds of their Sector.

Children and Bounds

Sectors are defined by their Renderer components and those of their children (i.e. the objects parented underneath them). In the case that a Sector has a child that is a Member, the Sector will ignore that child and all of its children for the purposes of computing the Sector's. The basic idea is that if a Sector finds that one of its children is a Member, then it ignores it, and assumes that child Member can and will take care of itself. The base Member will still act as a parent in every other way, in keeping with Unity conventions. This behavior is very useful if you want to have objects that are "part of" the Sector (like lights or particle systems) but which need to move within that sector, or change their bounds dynamically, or extend outside the bounds of that Sector.

See the Optimization chapter for information about how to reduce the CPU cost of bounds computation.

"Shared" Children

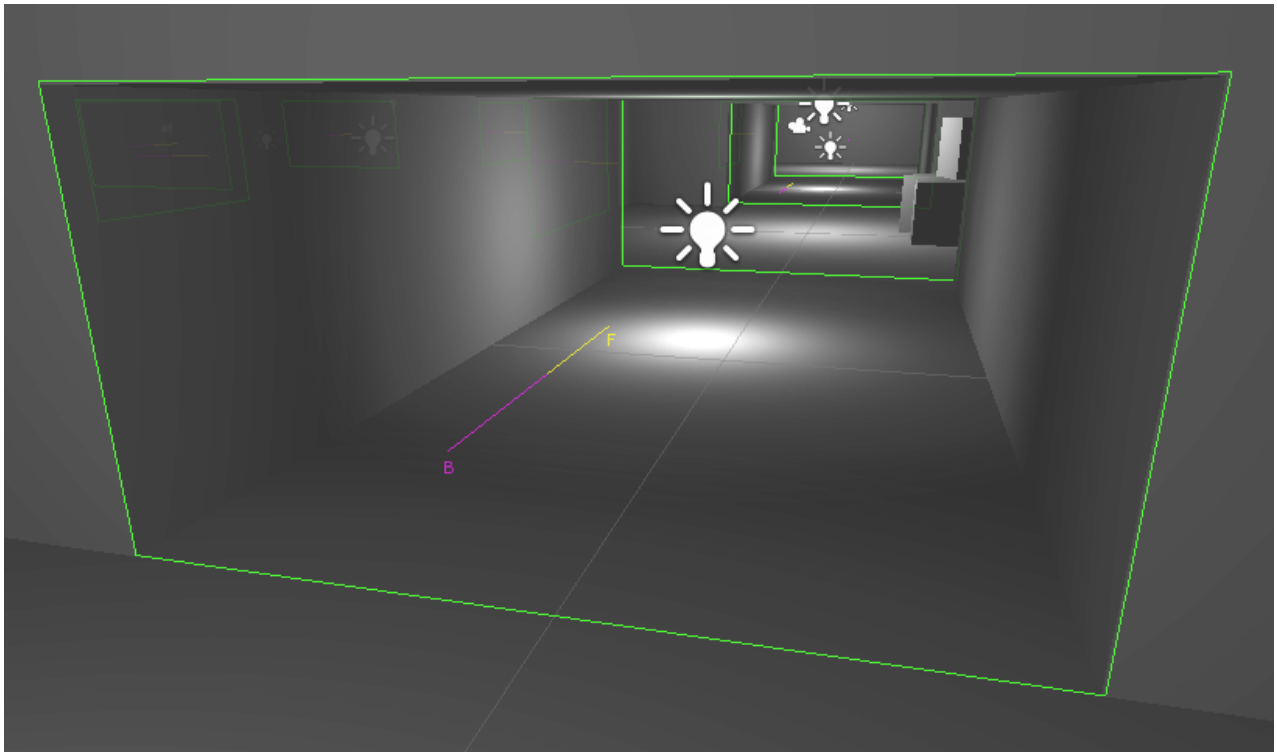
Because the bounds of a Sector are defined by the its Renderers, it's possible for some children (like Lights) to extend beyond the bounds of their parent Sector and overlap other Sectors. This may or may not be a problem, depending on the specifics of your scene. To help highlight these items, if you select a Sector and with Sector Gizmos enabled, any "shared" children will be highlighted in red. You can ignore them, fix them yourself, or press the Fix Shared Children button in the inspector. If you press the Fix Shared Children button, each shared child will be given a SECTR Member component. In general, though, you don't need to do anything unless you see a problem when testing your scene.

Membership

Sector membership is not exclusive. Members may be in multiple Sectors at once, so Sectors may overlap and even be nested in one another. Sectors should never be parented to one another. If this happens some assumptions will break, and strange things may begin to happen, especially in SECTR Stream and SECTR Vis. All of the SECTR editor UI will enforce this conventions, so just be careful when manually adding Sector components.

Portals

Portals represent the connections between Sectors. If a Sector is a room, then a Portal is like a doorway or window. Like a doorway, Portals have geometry that defines their shape, though in some applications the actual geometry is not strictly necessary. Because they connect Sectors, they should never be parented to a Sector, but be left at the same level of the scene hierarchy as the Sectors that they connect.



Connections

The most important thing that Portals do is connect two Sectors to one another. To do this, each Portal has two properties: a Front and a Back Sector. When both properties are filled out, the Portal forms a line between those two Sectors.

For many applications, it's important to know which side of the Portal the Sector is on, which is why the attributes are named as they are. The in-editor visualization will show you where the front and back sides are, but they are generally +Z for front and -Z for back, and the normal of the portal always points forward. When connecting Portals its important to get the sides correct, but if you get them backwards, you can simply press the "Swap Sectors" button.

Geometry

The shape of a Portal is defined by a Mesh resource. This is to allow users to create Portals both within Unity (using the included Portal drawing tools) and in external programs like Max or Maya. Portals are required to be planar (i.e. flat) and to be convex. They can, however, have as many sides as necessary.

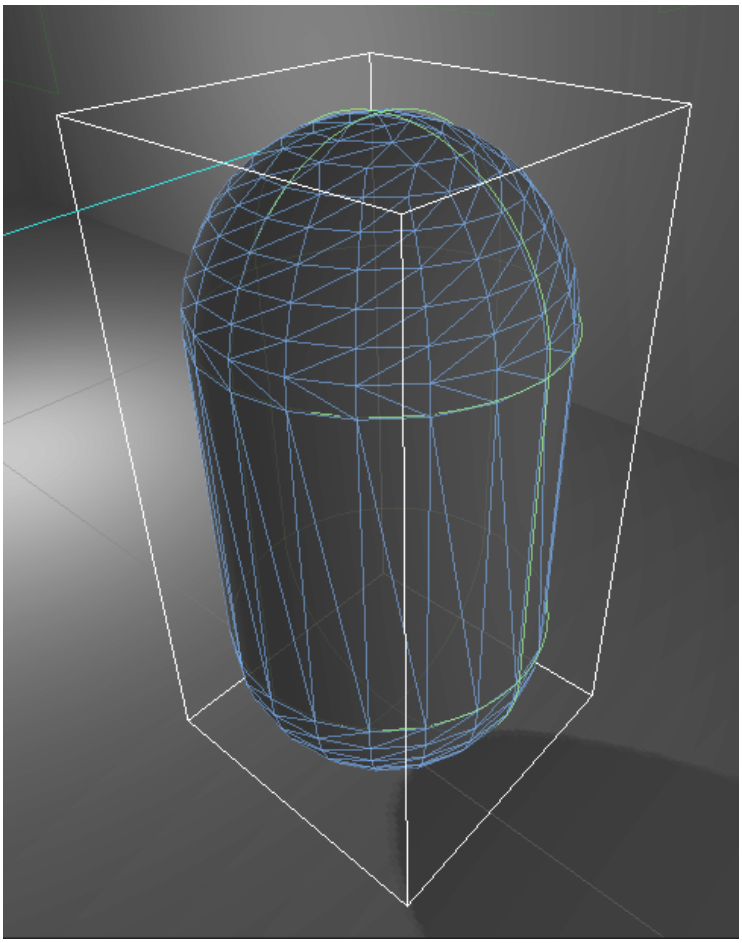
For most applications, Portal geometry does not need to perfectly fit the visual geometry of the level. Portals can almost always be left a simple shapes that extend a bit past the "real" opening that players see. As long as the Portal geometry is fairly close, everything will work fine. This is important, because some operations require more CPU the more sides a Portal has, so generally use the fewest number of sides necessary to accomplish the goal.

Flags

As the connections between Sectors, it's often useful for Portals to have some state associated with them, like being Open or Closed. SECTR provides some built in flags, but you are encouraged to add your own based on the needs of your game. All of the Sector/Graph algorithms will work with any flags you add, as well as the standard flags.

Members

While Sectors and Portals define the spaces and their connections, most objects in a game are neither Sectors nor Portals. These game objects often want to know which Sector(s) they are currently in, where the nearest portals are, etc. The Member component is designed to meet this need, keeping track every frame of which Sectors its a part of, and publishing that information to anyone who wants to know about it.



Useful Applications

Members serve many useful roles in the different SECTR modules, but the most basic idea is that if an object needs to know about Sectors but is not itself a Sector, then it should have a Member component added to it. Many components in other SECTR modules recognize this, and will add Member components for you automatically.

Children and Bounds

Like Sectors, Members can have children and their bounds are defined by the components in their children. Unlike Sectors, the bounds of Members are the union of all Renderer and all Light components, which ensures that any visual influence they have is fully represented by their bounds.

In the case that a Member has a child that is also a Member, the base member will ignore that child and all of its children for the purposes of computing its bounds, Sector memberships, etc. The basic idea is that if a Member finds that one of its children is also a Member, then it ignores it, and assumes that child Member can and will take care of itself. The base Member will still act as a parent in every other way, in keeping with Unity conventions.

See the Optimization chapter for information about how to reduce the CPU cost of bounds computation.

Static and Dynamic

Like Sectors, members may be static or dynamic, and may or may not have children. If a Member is static, it will save some CPU time and do fewer calculations each frame. If a Member is static, all of its children should be static too. If a static Member has dynamic children, the Member may not give the right information about which Sectors it's in.

Membership

Sector membership is not exclusive. Members can handle being in multiple Sectors at once, and every system in SECTR that needs a Member is designed to work with multiple Sector membership.

By default, Members are included in every Sector whose Bounds overlap with the Member Bounds. While simple and fast, this approach does not work well for some games with complex, interior geometry. For games with a significant amount of Sector overlap, nesting, or convexity, see the Portal Determined Membership section below.

Portal Determined Membership

As described above, by default Member components belong to all of the Sectors whose Bounds they overlap. For some games with complex scenes, this behavior may be undesirable, as Members may be part of too many Sectors. One simple example of this is a room nested inside another room. Some games may want the Member to only be part of the inner or the outer room, not both (which would be the case by default when the player is in the inner room).

SECTR provides a solution to this problem with the Portal Determined flag. This flag changes the membership computation so that it only changes when a Member passes through the Portal leading from one Sector to another. When the Member's transform position passes through the Portal geometry, the Member will leave the old Sector and enter the new one.

To support this feature, Member also includes a Force Start Sector feature, which allows you to specific a specific Sector to be used when the Member is first created/enabled. If not specified, the initial membership will be determined by the default bounds logic. Force Start Sector is useful for cases where the default behavior is undesirable, for example if the Member started in the inner room described above.

If using Portal Determined Membership, remember that the change happens when the transform position passes through the Portal, not the Member bounds. If your game has Members whose object position is at the bottom of the object, make sure that all of your portals extend at least a little bit below the floor so that the Member transform does not accidentally "slip under" the portal due to numerical precision issues.

Core Libraries

In addition to the components and tools described above, SECTR includes libraries that perform basic functions related to Sectors and Portals. These routines are foundational to many SECTR algorithms, and are used in CORE and the other modules.

Sector/Portal Graph

When Sectors and connected by Portals, they form a graph, called the Sector/Portal Graph. In SECTR we think of the Sectors as nodes of the graph, and Portals as the edges between them. The Graph library includes many useful functions for easily exploring the graph including:

- **Relationships**: This includes information on how the Sectors and Portals are connected, like which Sectors are the neighbors of the current sector, or how many Portals there are between one sector and another.
- **Traversal**: The Graph can be traversed in a variety of ways. The Sector/Portal graph is cyclic, so some care must be taken to not-revisit nodes unless desired, which the included traversal routines demonstrate.
- **Pathfinding**: Should you want to find the shortest path between two points, the Graph includes pathfinding routines, which are based on a well optimized A* based pathfinding routine that should be sufficient for most needs.

Geometry Routines

SECTR is, at its core, a spatial library. Much of its value comes from doing interesting things in 3D (or 2D) space. Many of these routines require access to a common suite of geometric functions, which SECTR CORE provides through the SECTR_Geometry library including:

- **Spatial Queries:** Determine which Sectors contain a particular point or volume.
- **Bounds:** Compute the extends of different kinds of objects (like lights) and evaluate their intersections.
- **Geometry:** Evaluate meshes for planarity, convexity, and the like.

Optimization

SECTR CORE is designed to be easy and correct by default, while also being as efficient as possible. As you become more familiar with SECTR and how you use it in your game, you can use the following techniques to further reduce the CPU cost of SECTR components.

Marking Objects as Static

If you know that an object like a Sector, Portal, or Member will not move during gameplay, you should mark it as Static. When marked as Static, SECTR objects will precompute as much data as possible on Start, saving CPU time during regular gameplay. SECTR components on static objects can be safely enabled, disabled, created, and destroyed during gameplay.

Using Member Bounds Modes

In order to work correctly, Sectors and Members need to compute information about their children and their bounds. For games with a large number of non-static Sectors or Members, this CPU time can add up. However, you can significantly reduce the amount of time spent in SECTR_Member.LateUpdate by using the Bounds Update Mode attribute.

- Static: Makes the object behave as if its Game Object was marked static. Use this when marking the entire object as static causes problems.
- Always: Updates all children and bounding information every frame. This will always give the correct results for dynamic objects, but it's the most expensive, and is often overkill.
- Movement: Updates all children and bounding information when the object moves, but does nothing while the object is stationary. Much cheaper than Always for objects that move around periodically, but are mostly stationary.
- Start: A hybrid of Static and Movement, this computes the children only at Start and updates the bounds when that object moves. This is the fastest Bounds Update Mode, and is ideal for objects that are always on the move, like NPCs, but whose children don't change significantly during their lifetime.

Advanced Bounds Controls

Much of the behavior Sector and Member are determined by their Bounds. SECTR does everything it can to efficiently compute the correct, most useful Bounds by default, but sometimes you simply need more control. SECTR provides the following controls for refining or overriding the default Bounds logic.

Extra Bounds

Extra bounds allows you to specify a certain amount of extra "padding" to the Bounds of a Sector or a Member. SECTR uses a small amount of padding by default to work around numerical precision issues, but you can use more (or less) if you want to quickly grow or shrink the default computed bounds.

Bounds Override

There are occasions where the default computed bounds are simply insufficient and need to be overridden. In these cases, you can specify your own bounds by ticking the Bounds Override check box and entering in your own bounding volume. Note that the bounding volume is in world space and will not update if the Sector transform is moved around.

Bounds Update Mode

Member includes a number of algorithms for how and when the recompute the Bounds, algorithms that trade accuracy for efficiency. These options are described in greater detail in the Optimization section.