Read Me

This is the Space Graphics Toolkit (SGT) **Universe** feature. This contains components allowing you to create and manage a floating origin system, allowing you to create VERY large scenes in Unity (e.g. the size of the observable universe).

What is a Floating Origin System?

Normal Unity scenes store positions using three floats (Vector3), where a position of 0,0,0 is called the origin of the scene. This system works great for most games that take place in a small area, but space games often need to take place over thousands or millions of kilometers, which the Vector3 alone is simply not capable of doing consistently.

The issue is that float values lose decimal precision the higher they get (or lower for negative), which means if your spaceship is 1 million meters/units from the center of the scene, the accuracy of the camera and position values will be very poor, giving you graphical issues, physics issues, and so on. So you want to keep your position values as close to the origin (0,0,0) as possible, so float accuracy remains high.

One solution is instead of moving your camera, you keep your camera fixed at the 0,0,0 origin, and move all the objects in your scene by the distance the camera would have normally moved. This works great, however, it can result in performance issues, because moving objects can sometimes be computationally expensive (e.g. moving physics objects). It can also make scene setup more difficult, and may introduce issues with existing components and scripts that expect the camera to be able to move.

A better solution that SGT uses is instead of keeping the camera at 0,0,0 all the time, it snaps it back to 0,0,0 when it moves too far away. This approach gives good performance because objects only need to have their positions shifted occasionally, it also maintains good rendering precision because values are always near the origin.

Warning!

Before you proceed to use this floating origin system, keep in mind that updating an existing project to include it is difficult, because many things may break without it being clear what the cause is. I highly recommend you only include this in a new project, where you can add things one by one so you immediately know what change caused your scene to break.

Additionally, keep in mind that storing world space positions is not a good idea with a floating origin system, because those positions aren't persistent. For example, if you have a missile script that is chasing a specific Vector3 world space position, it will begin chasing the wrong position as soon as the camera snaps. To update these values you can hook into **SgtFloatingCamera.OnSnap**, you can also instead store relative world-space positions, or use a **Transform** component.

Finally, if you're making a game with multiple separated cameras (e.g. split screen, security camera), all cameras must be within a reasonable distance of each other at all times, because this floating origin system was designed with one observer in mind, so the origin itself can only move around one main camera at a time. This means you cannot have a split screen game where both players are on opposite sides of the universe, because it makes the scene setup simply too complex to handle.

Basic Setup

At the most basic level, you just need to add the **SgtFloatingCamera** camera to your camera GameObject (e.g. Main Camera), and the **SgtFloatingObject** component to all your other GameObjects (e.g. spaceship, planet, character, enemy, asteroid).

Once done, you will see that when your camera moves more than (default) 100 units away from the origin, its position snaps back to the origin. All floating objects in your scene will also snap back based on the snap distance.

NOTE: If you add the **SgtFloatingObject** component to a GameObject, if it has any parents, make sure none of them also have the **SgtFloatingObject** component. If they do, they will both snap, compounding the effect for the child. You should keep all floating objects separate.

Handling Snaps

If you need to update a script whenever the camera snaps, you can hook into the **SgtFloatingCamera.OnSnap** event. This event tells you which camera snapped, and how many world space units it snapped.

You can also hook into the **OnSnap** event from the **SgtFloatingObject** component.

High Precision Positions

The basic camera and object setup described above works well for very large scenes (e.g. millions of kilometers), but because the positions are still stored in the Transform.position values, it's insufficient if you need scenes that are gigantic (e.g. millions of lightyears). If you need scenes this large or even up to the size of the observable universe we live in, then you need to use high precision positions.

To enable this, tick the **Use Origin** setting in your **SgtFloatingCamera** component, this should add a new "**Floating Origin**" GameObject to your scene.

Next, add the **SgtFloatingPoint** component alongside all your **SgtFloatingObject** components.

This **SgtFloatingPoint** component used by both the camera and your objects stores a high precision universal position. The position is stored using Local X/Y/Z and Global X/Y/Z values. The local values are equivalent to unity world space units (meters), and the global values are equivalent to 50000000 units (meters) each. Each time a local value exceeds this 50000000 value, it will reset and increment the global value. When combined and used to their maximum, this allows them to accurately store positions of +- 461168601842738790400000000 meters, which is +- 48745559677 light years, or 4.87 billion light years, which is greater than the 46.6 billion light year radius of the observable universe.

NOTE: Using the high precision origin for the camera introduces the limitation that you cannot parent the camera to another GameObject, unless the parent is just for organizational purposes (e.g. it doesn't move or rotate). If you need to parent the camera to something, I recommend you use the **SgtFollow** component on your camera, this can be setup to follow another GameObject, with a given offset. The reason for this limitation is that the camera is now driven by the **SgtFloatingPoint** and **SgtFloatingOrigin** components,