Turbo Slicer Guide

Turbo Slicer version 1.4

Hi, Toby here from Noble Muffins.

This here's a slicing kit. It takes an object featuring a mesh, then cuts it in half (yielding two new objects). There's a few things you can do with it, a few things you need to know and a lot you probably don't but I'll put here just in case.

This package includes a pair of demos; one's a conventional touch-slicer game (with multitouch support!) while the other shows the blade kit. I've also included a glowy trails effect which, like the slicer demo, supports multitouch.

You can call down a slice directly with the Turbo Slice API or use the Slicer & Sliceable system, both explained in this manual.

One last thing; **please don't turn on that mesh cache thing until you've read about it in the manual!** Its behavior's a little esoteric but if you learn what it does or leave it off, you'll be fine.

Good luck!

Acknowledgments

John Ratcliff, a software engineer at NVIDIA, wrote the basic Plane-Triangle split in C++ and his code can be found here: http://codesuppository.blogspot.com/2006/03/plane-triangle-splitting.html

This kit began as a translation of his code into C#, but was heavily reworked to handle meshes, repeatedly slicing and to avoid cache thrashing.

Some of the demo code was written by fellow noble muffins Jack Dog (the project lead on Synergy Blade) and Toby's Muffin Apprentice (real name withheld).

Before You Start

You can access Turbo Slicer via the static property **TurboSlice.instance**. An instance in the scene will be created if one does not already exist. You may also create the instance yourself by adding the TurboSlice component to a game object. Do not manually add more than one; this will invoke undefined behavior.

You must create an instance manually if you want to configure this instance. The instance's "mesh cache" option is the only non-deprecated configuration option and it is described later in this document.

You will also find an "infill" property on the component; do not touch this if you're not already using it. This is an old interface for configuring the infill which is maintained, but not recommended.

Preparing an object

To slice an object, Turbo Slicer needs to be able to find a single **MeshFilter** and **MeshRenderer** in the target object's hierarchy. It does **not** support SkinnedMeshes. It **does** support meshes with multiple materials.

If you feed an object meeting the above requirements directly to Turbo Slicer via the APIs described later in this document, it will slice. However to configure the Turbo Slicer's behavior, you need to add the **Sliceable** component to this object. When you feed an object to Turbo Slicer, it will try to find a single Sliceable component either on it or in its children and use the configuration described there.

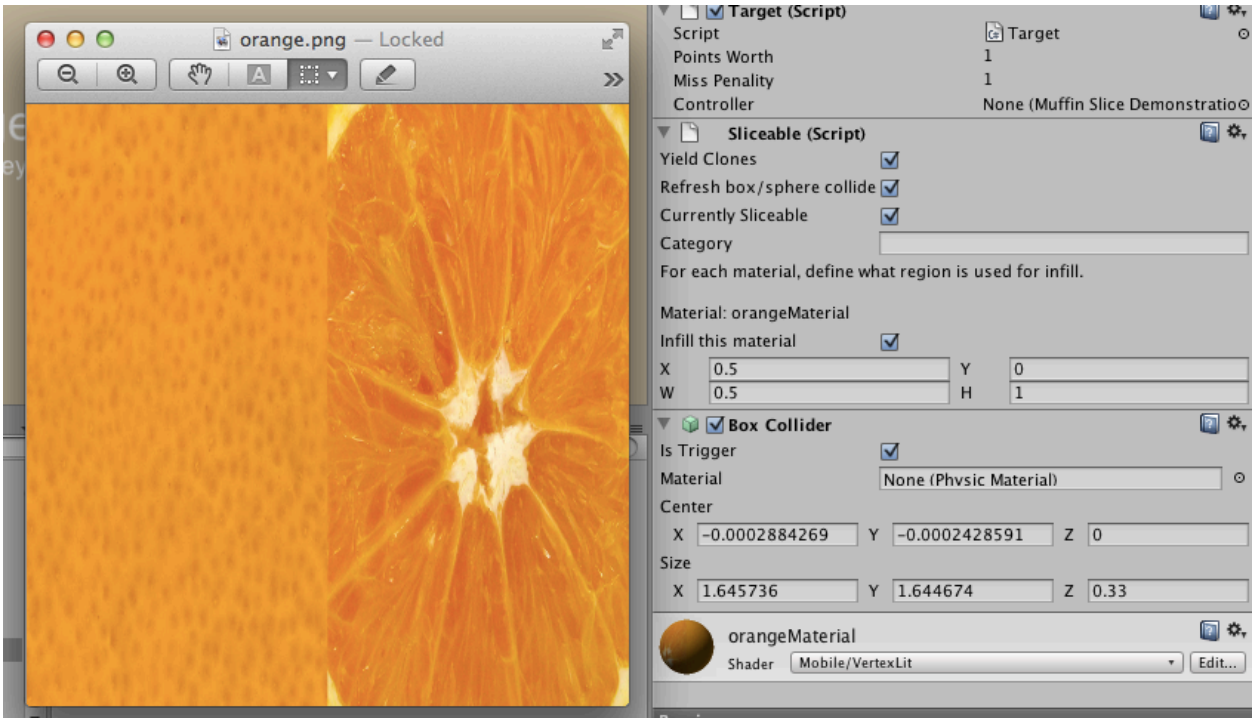
You may also extend the abstract class **AbstractSliceHandler** to create a component that will be called upon after a slice occurs to process the resultant objects. Place your component **next to** the Sliceable (same object).

One last point before we go through the configuration options in detail; giving an object hierarchy to Turbo Slicer that possesses multiple MeshFilters, MeshRenderers or Sliceables will yield undefined behavior. (However you **may** add multiple slice handlers.)

Infill

The hole made by the slice can be filled in with textured material, with a few requirements. The slicer will look at the **cross section** of the slice and try to find one or more **closed polygons**. An object like a plane does not give a cross section featuring any closed polygons. A ball, torus or any whole, **closed** object will.

The slicer will need to use **atlassing**. This means that if a given object uses material M, than the infill will also be done with material M. You must specify the region used as infill source data on a **per-material** in the **Sliceable** component, as shown here:



The materials present in an object will be shown in the Sliceable component. In this example, the material *orangeMaterial* has texture for an infill covering its right half. In the Sliceable configuration shown here, we see the region (0.5, 0, 0.5, 1) configured.

The left half of the texture depicts the exterior, and the input mesh's UVs map to this region. The right half depicts the interior, and the generated infill's UVs will map to this region.

If your object possesses multiple materials, you will need to configure them on a material-by-material basis. However multiple materials in an object is not recommended for infills unless each material maps to a distinct, closed whole.

You can see this example in action by running the included Slice By Line Demo.

Yielding Clones

When a given object A is passed to the Turbo Slicer, it will create objects B and C which have each half of the slice result. There are two ways we can create these objects; we can create fresh primitives or clone the source object.

In the former case, objects B and C will be instantiated as empties and configured to include the meshes. You might use the slice handler facility to configure them or simply process the slice results returned by the API.

In the latter case – with Yield Clones turned on – it will clone object A to create B and C. This way, Unity will take care of cloning all of object A's hierarchy (if present), components and component configurations.

When "yield clones" is turned on, another option will appear; Refresh Box/Sphere Collider.

Warning! Because the clones are newly instantiated, Unity will call Start and Awake. Consider this if you encounter unexpected behaviors.

Clone from Alternate Prefab

When yield clones is enabled, Turbo Slicer creates the slice results by cloning the source object. But it can also clone from a different prefab altogether.

Suppose you have an object from prefab A, and a matched prefab B. This matched prefab has a matching hierarchy, but perhaps has different code attached or a different material.

Now suppose we want the slice results to be based on prefab B, even though we are slicing an object based on A.

The Sliceable component lets us assign an **alternate prefab** if **yield clones** are enabled. It is then possible to use the alternative prefab to create child objects instead of the original object. You can explicitly tell it to always use the alternate prefab (a checkbox will appear) or decide when to use the alternate prefab by extending the **Abstract Slice Handler** as described below.

Refresh Collider

This here's a convenience. When you (for example) add a BoxCollider to an object, Unity will automatically fit the collider to the object. If you add a BoxCollider to an object at runtime, this will happen; it will be sized automatically.

However, when we clone to create the new objects, **if** the source object had a box or sphere collider, it will retain the old size configuration which is now inappropriate.

Mesh colliders fail as well. One might expect them to defer to the mesh filter and adopt the object's new shape, but they do not; they retain a direct reference to the old, unsliced mesh.

This feature if turned on will instruct Turbo Slicer to refresh any xox, sphere or mesh colliders if present in object A. That way, the colliders – if present at all – will be correctly sized for each half of the slice result.

Currently Sliceable, Category

These are used by the Slicer facility, described later in this document.

Abstract Slice Handler

This is not part of Sliceable, but it is an abstract class which you can extend to customize the slice process. When you create a slice handler with your custom code, add it to the sliced object next to the Sliceable component. You may add multiple slice handlers to an object, but keep in mind that there is no guaranteed order in which they will be called. There are two methods you can override; **handleSlice** and **cloneAlternate**.

To process slice results after a slice, create a component that inherits from the abstract class **AbstractSliceHandler** and add this method:

```
public override void handleSlice( GameObject[] results ) {
   // ...
}
```

When a slice occurs – and before the original object is destroyed – you are given the opportunity to affect both slice results based on the sliced object's state and your own procedures. For an example, refer to the provided **DemoSliceHandler** which uses explosive force to blow them apart from each other. If your object A is sliced into objects B and C, and you want to perform some setup on B and C based on the state of A, you may do so here.

Meanwhile, the clone alternate method permits you to decide if a given slice half will be based on the original object or the alternate prefab (described prior in this document).

```
public virtual bool cloneAlternate ( Dictionary<string,bool>
   hierarchyPresence ) {
   bool useAlternatePrefab;
   // ...
   return useAlternatePrefab;
}
```

When a slice occurs, for each half this method will be called with a dictionary describing what child objects in the hierarchy are present. Suppose the main slice object is a live character and alternate prefab represents a dead character; we might come up with a list of vital child objects (head, torso, et cetera) and decide that if they're not all present, we want to use the dead prefab instead. We would do this by verifying the presence of all in the list and returning true only if all are accounted for.

Performance

Who is this for?

You need to read this section if you plan to release a product on mobiles. On a desktop or laptop – or for low-pressure projects – feel free to skip this but if you need shine on mobiles than you'll need to read this carefully.

Infill

The infill feature added in version 1.2 is in any early state. It works – most of the time – but it is not as optimized as it could be yet, and does involve a theoretically large amount of computation.

If it performs poorly for you on a mobile, consider using the included double-sided shader to give an impression of different insides.

General Considerations

Turbo Slicer is a heavily **CPU bound** operation. It is heavily optimized (albeit while remaining platform independent) but in the end all operations take time to work. Turbo Slicer was build to permit the game Synergy Blade to run at 60 FPS on iPad 1 while slicing repeatedly, without causing lurches. However to say “Turbo Slicer goes at 60 FPS” is a bit simplistic.

To reach 60 FPS, your game must have the frame ready & delivered to the screen in about **16 milliseconds**, and **everything** you ask it to do eats into that time budget. To accomplish 60 FPS without visible lurches, we had the game produce a frame in a bit under 16 milliseconds, so that there was still enough free time to add a slice every so often without going over.

It did need to be very fast to fit in the margin, and this current version is approx. 30% faster than what was released with Synergy Blade. But the basics facts remain; **adding work adds time** and **to avoid a lurch, you need to make time for it**.

What adds work is **geometry**. Every triangle & vertex is work for it. To meet the iPad 1, 60 FPS target we kept the models under 400 or so triangles. A newer iPad can handle more, a PC or laptop can handle **vastly** more and if you target instead 30 FPS (which is legitimate) than you can give it a lot more geometry.

Turbo Slicer is roughly **O(N)** with geometry load. (Look up “Big-O notation” if you’re not familiar with this.)

Mesh Caching

In Synergy Blade, players would slice targets **repeatedly** and the kit has a cache system designed specifically for this. **This feature assumes that sliced elements will fall away from the screen and be destroyed**. If this is not true for your game, do not turn on this feature.

The core idea is that there is a cost associated with pulling geometry from a Unity mesh and into the slice kit. With caching enabled, if you slice object A into objects B and C, than Turbo Slicer will retain the geometry data for objects B and C. This means that when you slice object B into new objects D and E, it will have the data ready to go.

Furthermore, objects B, C, D and E (all resulting from slices) will all be associated with the **same** internal mesh cache, X. This cache was created when the original mesh A was sliced, and is large enough to accommodate a very large number of subsequent slices. (In Synergy Blade one might slice an object up to 30 times.)

Only when **all** objects in the family (B, C, D, E, etc.) fall away from the screen and mesh cache X is no longer associated with any living object will mesh cache X be dereferenced. (That is, abandoned for garbage collection.)

If the feature's turned on inappropriately, there is a risk of a memory leak, so for safety purposes the Turbo Slicer will automatically disable this feature and abandon all caches if it detects a cache lingering for more than 15 seconds. A warning will be delivered via Debug.LogWarning.

Garbage Collection

With or without mesh caching, all slices involve large heap allocations depending on the amount of geometry coming in, so a large number of slices over time **will** necessitate a garbage collection. I will share how we avoided visible lurches in Synergy Blade.

First, we used a script called the "HeapExpander" to fluff the margin between the game's normal operating memory usage and the collection trigger. This is to extend the amount of slicing that might go on before the first collection.

Second, the game was designed with **frequent breaks**. Bosses – this is a game where you slice bosses – would show up in groups, then frequently there would be a rest with no bosses at all, including transitions from one part of the office to another. At the start of each break, the game controller would explicitly call the garbage collector to restore the margins.

Both of these techniques are implemented in the included **Slice By Line Demo**, described later in this document.

Normals

A given mesh has multiple data channels that need to be processed. Every data channel creates work, but not every one is needed. Specifically, normals can be disabled.

To avoid branching in hot code or unoptimal code organization, these must be switched at **compile time** via preprocessor directive. But don't worry, it's a snap. Open TurboSlice.cs and look at the very top. Instructions are there.

Normals are **on** by default.

Switchable support for the color channel and secondary UV channel will be added in a future release.

Turbo Slicer API

There are five public APIs. They are not static.

```
GameObject[] splitByLine(GameObject target, Camera camera, Vector3 start,
   Vector3 end)
GameObject[] splitByLine(GameObject target, Camera camera, Vector3 start,
   Vector3 end, bool destroyOriginal)
GameObject[] splitByTriangle(GameObject target, Vector3[]
   triangleInWorldSpace, bool destroyOriginal)
```

```
GameObject[] splitByPlane(GameObject target, Vector4 plane, bool
   destroyOriginal)
GameObject[] shatter(GameObject go, int steps)
```

Each of these takes a given GameObject conforming to the specifications laid out earlier in this document (See: **Object Requirements**). Each returns an array containing either the **given object** (if not sliced) or **two resultant objects**. (Shatter may give more than two.) These new objects will be positioned & rotated to match their forebear. Lastly, the **destroyOriginal** parameter specifies if the Turbo Slicer should automatically delete the given game object (if sliced). (Shatter and the first splitByLine without this parameter assumes the value **true**.)

Split by Line

Split by line will try to slice the object in screen space. Given a start and end coordinate on the screen (such as from Input.mousePosition) and assuming the player is looking through the given camera, it will split the mesh accordingly.

This is used in the Split By Line Demo described later in this document.

Split by Triangle

We can also describe a plane by providing three vertices that lie on that plane. This method takes three vertices in **world space** to perform the slice. (It does not need a camera as it does not need to make any perspective oriented transformations.)

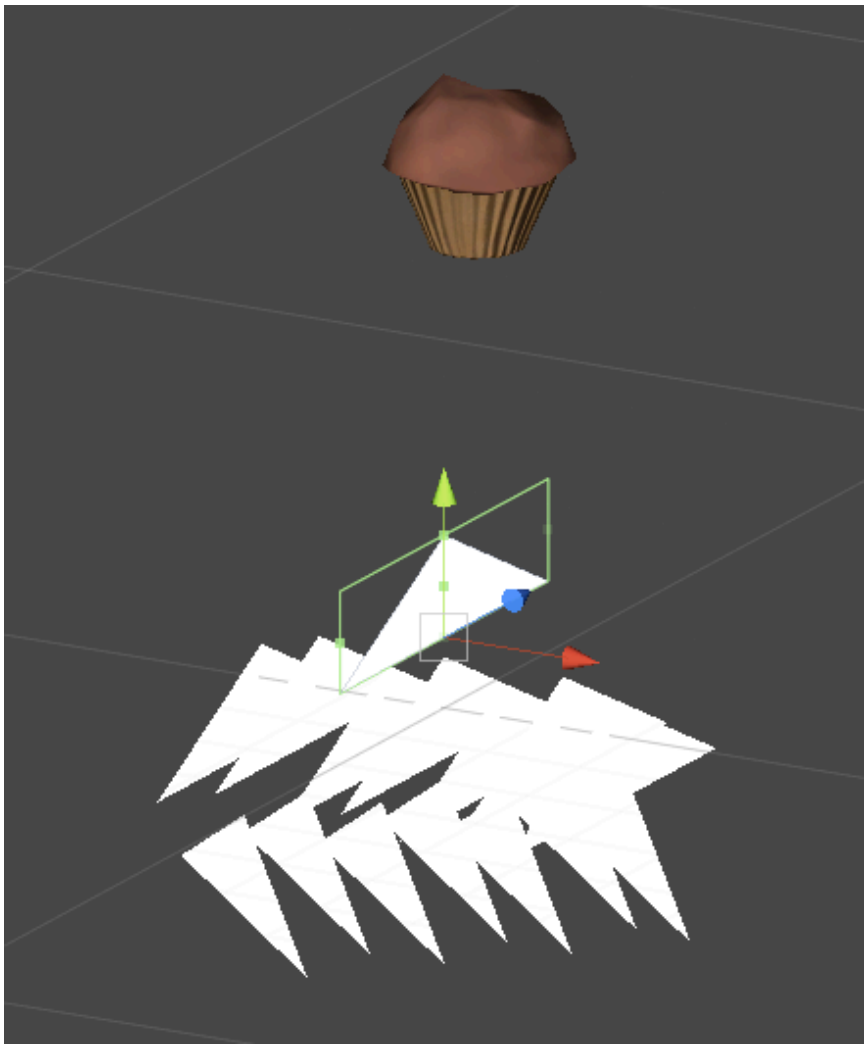
Split by Plane

Finally, you can slice using the **equation of a plane** in **local space**. The plane equation figures are given in a Vector4. Exactly how to work with these figures is beyond the scope of this document, but I've made this method public in case you're familiar with these figures and would like to use them directly.

Shatter

This performs random slices on an object a specified number of times and returns the set of objects it produced. It will destroy the original and all intermediate objects.

Slicer



The Slicer & Sliceable components are an easy way to make an in-game object cut another. There's two parts; the Slicer **prefab** and the Sliceable **component**.

Slicer

The Slicer prefab is a blade that cuts Sliceables.

You can rotate and size it, then place it in a hierarchy. You can move it at runtime. You might, for example, attach it to a sword. Remember that the white triangle will only appear in the editor; you do not need to manually hide it.

You can manually create an object for the scene's single TurboSlice component and connect it to the Slicer's "TurboSlice" field in the Inspector. However if the slicer isn't provided one, it will find it and if it can't find it, it will create it at runtime.

The prefab possesses a few empties for "Plane Definition"; their positions are fed to the Turbo Slicer's splitByTriangle method. It's not necessary or recommended to mess with any of this, however.

Sliceable

The Slicer will only automatically slice objects if they're configured for it. To do this, you must attach the Sliceable component to the object, with an additional specification; it must have RigidBody and Collider components as well. (The Slicer uses OnTriggerEnter and is thus subject to its requirements.)

If you want to **suppress sliceability**, your object controller can disable the Sliceable by setting the property **currentlySliceable** to false. The Slicer performs all operations in LateUpdate so it will observe this property **after** you've set it without waiting until the next frame.

For example, an enemy that can take multiple hits might leave currentlySliceable off until the final hit, at which point it's flipped on. If this is done in an Update, than the Slicer (which uses LateUpdate) will observe this and perform the slice on the same frame (assuming you use the same collider to detect hits that the Slicer does).

Slice Handler

Using the Slicer facility, you do not call the API and therefore do not have the slice results returned to you. Therefore to perform custom configuration on the slice results after the slice when using the Slicer, you must extend the Abstract Slice Handler as described earlier in this document.

Categories

Suppose you want sword A to only cut enemies, while sword B can only cut the hero. For this you may assign a category to any Sliceable, and specify that a Slicer may only slice for that (or multiple) categories.

On Colliders and Accuracy

The slice kit is not very efficient at determining if object A will or will not be sliced by plane P; it will process it with full accuracy every time. So to avoid work, it is appropriate use colliders to judge whether or not to *try*.

But suppose you have a collider that's a bit too large or simple; say a SphereCollider around a very non-spherical object. Suppose the collider's oversized bounds trigger a slice when it's not exactly appropriate to do so.

The result is that it will not be sliced; the slice method will perform a more accurate assessment and simply return the original object.

Therefore, it's OK to use a very simple collider for work avoidance; it will not harm accuracy in the end. It's absolutely not necessary to use a MeshCollider – at least not for this purpose.

Sliceable Demo

This can be found in Noble Muffins / Turbo Slicer / Sliceable Demo. It's a very simple demonstration of the Slicer / Sliceable construct; it drops a sliceable muffin onto a set of blades.

As mentioned in the section on Slice Handlers, it includes a demonstration of a slice handler which sets up the slice results for physics and further sliceability.

It also uses a "destructofloor" which zaps anything that goes under -10 units on the y axis.

Slice By Line Demo

This much more substantial demo can be found in Noble Muffins / Turbo Slicer / Slice By Line Demo. It is a partial implementation of a Fruit Ninja clone, but with muffins.

Some parts, such as the Spawner, are self explanatory while others, such as the Heap Expander, are explained elsewhere in this document. But the core of it's the **Demo Controller**.

The Demo Controller is responsible for handling mouse and multitouch input and calls the slices. (Note that it doesn't interact with the Slash Effect which acts independently and is described later in this document.) It also implements some primitive game rules which you may modify.

You might also add sound effects here. You can find the sounds we used for Synergy Blade in **Spearhead Sound Effects** on the Unity Asset Store.

If you trawl through the code in MuffinSliceDemonstration, you can see comments showing where to edit the code to add what you want, and what existing code you can utilize for what.

Remember to clone the scripts before you edit them to prevent them being overwritten by an update.

Go nuts.

Double-Sided Shaders

There are four simple shaders included which have visible interiors. Two of them support lighting while two do not, and two of them support separate interior, exterior textures while the others do not.

In Synergy Blade, we used the two-texture, unlit double-sided texture to paint the bosses; they would have a boss texture outside, with a "meat" texture inside (found on cgtextures.com).

This kit is used on the muffin material in the Sliceable demo.

Slash Effect

Dropping the Slash Kit prefab into a scene activates the slash kit.

Important! You'll want to either move this somewhere far out of the way or use the layers and rendering settings to make it not interfere with other elements. Unfortunately we cannot export layers, so you will need to do this on your end.

Contact

If you run into any problems at all, let me know at toby@noblemuffins.com.