Turbo Slicer Guide

Turbo Slicer version 1.6

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.

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

We've made 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 find them here.

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.

A vector-vector transformation algorithm used is pulled from a 1992 forum post by a Ben Zhu who worked for SGI at the time. The thread can be found here: http://steve.hollasch.net/cgindex/math/rotvecs.html

Before You Start

You can access Turbo Slicer via the static property **TurboSlice.instance**. You do not need to manually create an instance; one will be created automatically.

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.

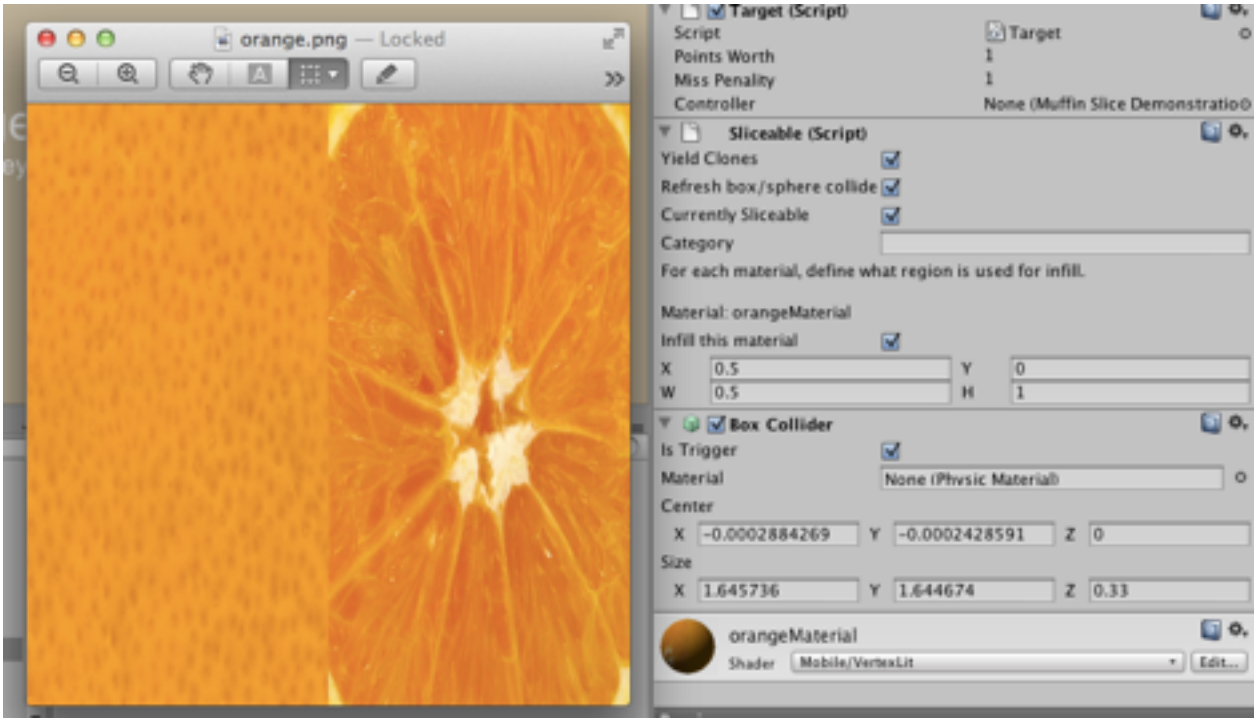
You may also extend the abstract class **AbstractSliceHandler** to create a component that will be called upon after a slice occurs to process the resultant objects. Place your component **next to** the Sliceable (same object).

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 a texture **atlas**. 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. It will create these objects by **cloning** object A (including its hierarchy) to yield objects B and C, **applying** the new meshes to these new objects then **deleting** the original object.

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

Clone from Alternate Prefab

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 scripts attached or a different material or configuration.

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**. 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 will instruct Turbo Slicer to refresh any box, sphere or mesh colliders. 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 demo's **TSD3DemoSliceHandler** 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. It 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

There was, in previous versions, some manual control over and feedback from the mesh caching. This is no longer the case; mesh caching is handled internally, will not disable itself, will not produce warnings and does not require any mental load for the developer.

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.

To hide lurches in Synergy Blade, the game was designed with **frequent breaks**. Bosses – this is a game where you slice bosses – would show up in groups. There would then be rests 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 margin between current heap mass and the trigger point.

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**, **tangents** and **UV2** can be toggled.

Add a Sliceable component to an object to permit editing its configuration and use the check boxes.

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.)

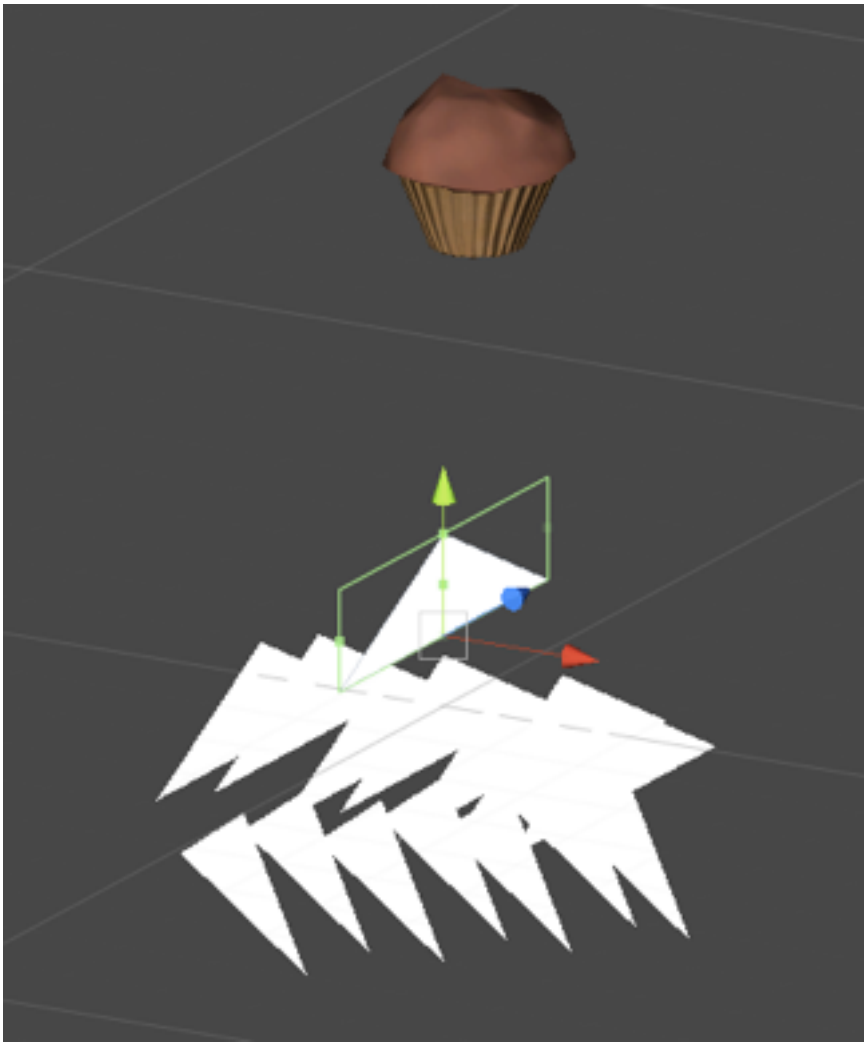
Split by Plane

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 employs both OnTriggerEnter and OnCollisionEnter and is thus subject to those constructs' requirements.)

If you want to suppress sliceability, your object controller can disable the Sliceable by setting the property **currentlySliceable** to false.

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).

The Slicer performs all operations in LateUpdate so it will observe these properties after you've set it without waiting until the next frame.

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.

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 Demos / Sword Demo TS.

Muffins a flung up into view where the player, wielding a mouse controlled sword, can slice them. At its core, it is nothing more than a controllable sword with a Slicer attached to that sword. The muffins, which have Sliceable instances, can be cut by the Slicer.

This demo is not the bare minimum; a number of scripts are included to “flesh out” its behavior and we’ll discuss here what they do so you dear user can utilize them for fun and profit.

The Sword

TSD3SwordTransformController is responsible for steering and turning the sword to chase the mouse cursor. It needs to know the relevant camera and the reach of the blade in order to transform mouse coordinates in screen space to a sensible-enough orientation in three space. The tracking time controls the speed at which the blade chases the cursor; lower is faster.

TSD3SwordVelocityFilter is responsible for answering a question: Is the sword moving fast enough to perform a slice? The length of the sword is given by the user because parsing the mesh isn’t reliable. The minimum tip speed for cutting is in units per second. It has one public boolean property: **IsFastEnoughToCut**. The next two components refer to this class and use this public property.

TSD3SwordSwishController observes TSD3SwordVelocityFilter and when it goes from too slow to fast enough, it will emit a swish using the adjacent AudioSource. It can be given a set of sound effects and play them at random.

TSD3AttachableSlicerController observes the TSD3SwordVelocityFilter and keeps a reference to the Slicer. It ensures that the Slicer is enabled or disabled depending on whether or not the sword is fast enough to slice.

Slice Handlers

TSD3SoundWhenSliced is an AbstractSliceHandler implementation. It must be attached to a Sliceable, and causes a sound to play when the object to which it's attached is sliced.

TSD3BurstApartWhenSliced is an AbstractSliceHandler implementation responsible for applying explosive force to pop the slice results apart.

Game Rules

TSD3ScoreModel is not a Unity component. Access TSD3ScoreModel.instance from a script to observe and manipulate the number of primary slices (when a whole object is sliced), secondary slices (when debris is sliced) and misses (sliceables which fall out of range).

TSD3ScoreAgent is a Unity component attached to sliceable items. It is responsible for pushing points and penalties onto the score model when a slice occurs. To do this, it keeps track of whether an object is fresh or if it is the result of a slice.

The View

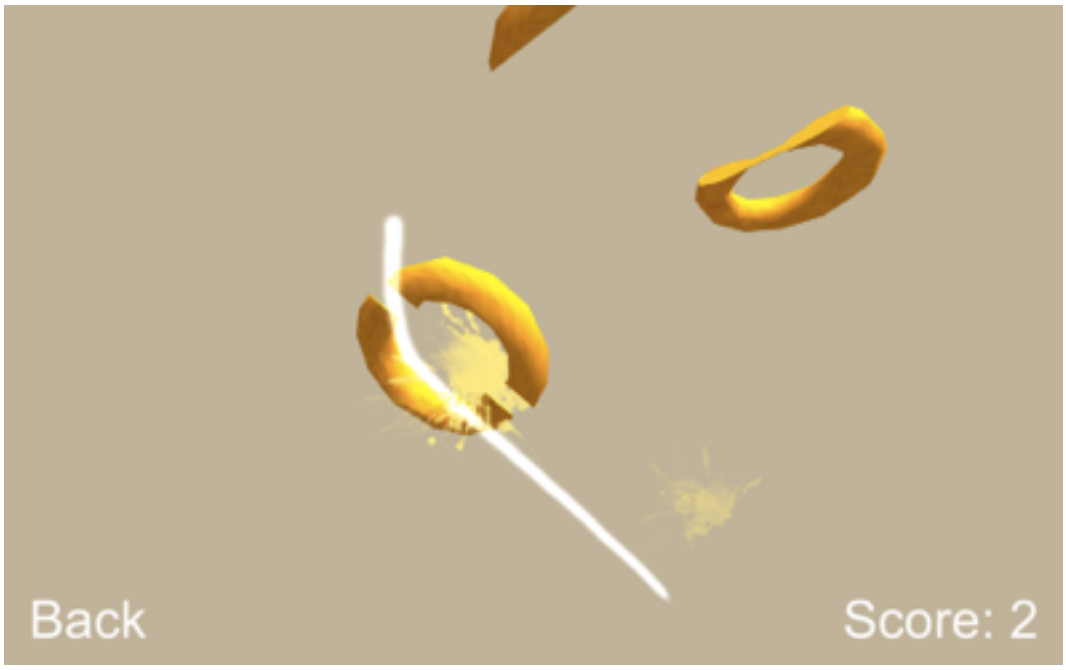
TSD3FailSounder observes the score model and makes a noise when a miss occurs.

TSD3ScoreView pushes the current score (from TSD3ScoreModel) onto a coresident text mesh.

The Spawner

TSD3Spawner throws objects upwards at the necessary speed to reach a given height (from the starting point) based on the current gravity.

Touch Slicer Demo



This demo is a rewrite of the older “Slice by Line” demo. It’s a partial implementation of a Fruit Ninja clone, but with “donut oranges” and has been rewritten from its earlier, messier form to mimic the decentralized, single-responsibility approach of the sword demo above. Let’s walk through its classes.

Touch Slicing

This is managed by two components; one on the targets responsible for identifying the targets to the slice core and one on an empty object responsible for calling for slices based on input.

TSD4Target is responsible for maintaining a list of all live targets on the screen. That’s all. The list is exposed through the static property TSD4Target.targets. This static property returns a read only list.

TSD4TargetSlicer is responsible for observing the mouse and touch inputs through Unity’s Input interface and slicing targets as appropriate. To work, it needs to know through what camera is the player looking at the targets (the public “camera” field).

If you have targets in the camera’s field of view, the TSD4TargetSlicer can be trusted to slice them when a finger or mouse cursor is dragged across that target.

It must be known that it does not use ray casting; rather it attempts to determine the target’s size on screen, and then performs a collision test in two-space. This gives it freedom to test multiple input samples on each target per frame without punishing lower end mobile CPUs.

Slice Handlers

TSD4SoundWhenSliced is an AbstractSliceHandler implementation. It must be attached to a Sliceable, and causes a sound to play when the object to which it's attached is sliced.

TSD4BurstApartWhenSliced is an AbstractSliceHandler implementation responsible for applying explosive force to pop the slice results apart.

Game Rules

TSD4ScoreModel is not a Unity component. Access TSD4ScoreModel.instance from a script to observe and manipulate the number of primary slices (when a whole object is sliced), secondary slices (when debris is sliced) and misses (sliceables which fall out of range).

TSD4ScoreAgent is a Unity component attached to sliceable items. It is responsible for pushing points and penalties onto the score model when a slice occurs. To do this, it keeps track of whether an object is fresh or if it is the result of a slice.

The View

TSD4FailSounder observes the score model and makes a noise when a miss occurs.

TSD4ScoreView pushes the current score (from TSD4ScoreModel) onto a coresident text mesh.

The Spawner

TSD4Spawner throws objects upwards at the necessary speed to reach a given height (from the starting point) based on the current gravity.

Trails Kit

Dropping the Trails Kit prefab into a scene activates the trails 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.

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.

Contact

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