Limb Hacker Guide

Limb Hacker version 1.0

Hi, Toby here from Noble Muffins.

This here is a slicing kit. You give it a character and a rag-doll version of that character, and it'll hack the limbs off. It can also hack off the torso and head, and cut anywhere along the bone.

This package includes a demo where you play as a sniper with a good sight on a militiaman with an AK.

You can slice from a particular joint with the Limb Hacker API or use the Sliceable-By-Point component to slice whatever's nearest to a point in three-space.

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 create Turbo Slicer and further reworked into this.

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

The demo's militiaman was created by artist Nigel Kitts.

Before You Start

You can access Limb Hacker via the static property **LimbHacker.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 LimbHacker component to a game object anywhere in the scene where you are using it.

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, Limb Hacker needs to be able to find a **SkinnedMeshRenderer**. It **does** support meshes with multiple materials.

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.

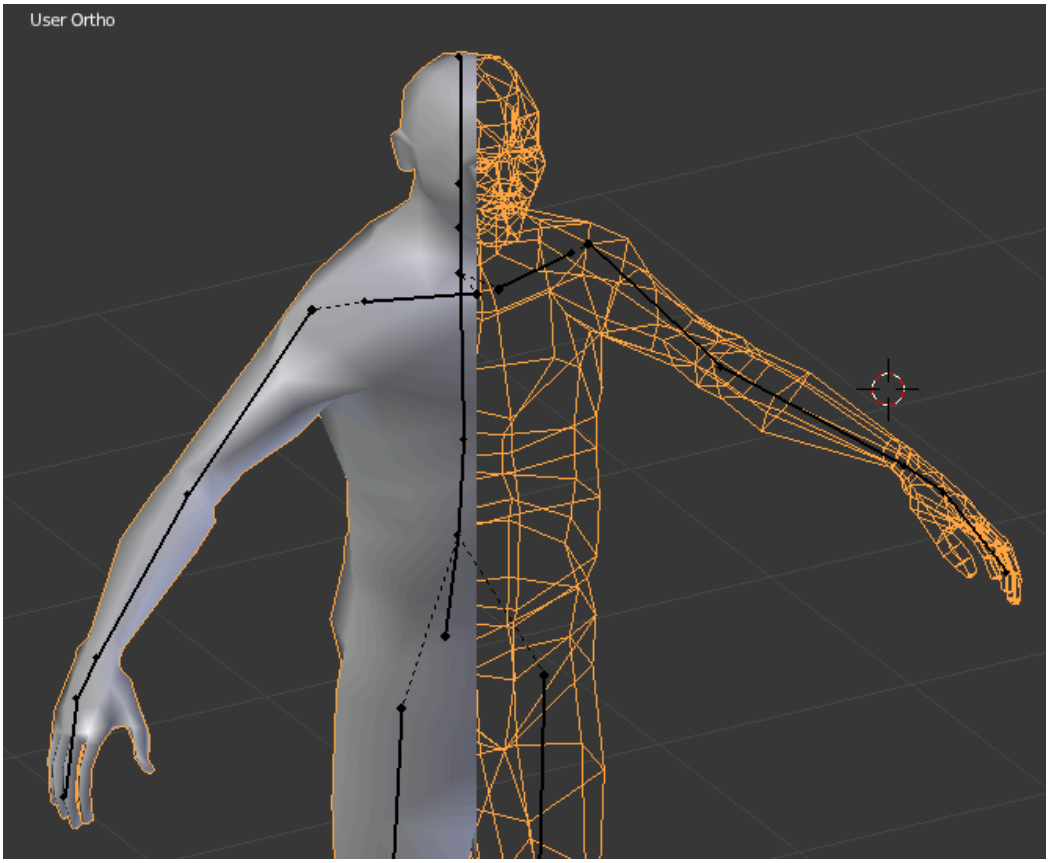
If you feed an object meeting the above requirements directly to Limb Hacker via the APIs described later in this document, it will slice. However to configure Limb Hacker's behavior, you need to add the **Sliceable** component to this object. When you feed an object to Limb Hacker, it will try to find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

If you wish to sever by points instead of by joint name, you'll need to add the **SliceableByPoint** component. This component keeps the severByPoint method's configuration; it needs a list of severable joints. This permits you to control which joints are *not* to be sliced, which is important because you may find some joints yield ugly results.

ToRagdollOrNot is a component responsible for determining whether or not a slice results in the entity becoming a ragdoll. You can use it (it is configurable) or write your own decider by extending the abstract class **AbstractSliceHandler**. Place your component next to the Sliceable (same object).

Ideal Mesh

The slicer is designed assuming it will be dealing with closed, textures meshes. Meshes which have layered, hidden geometry or triangles which pass through each other may result in infill fails. (The “infill” is the geometry made up to cover holes made by the slice.) We plan to improve this, however as of this release it is optimized for closed meshes like this one:

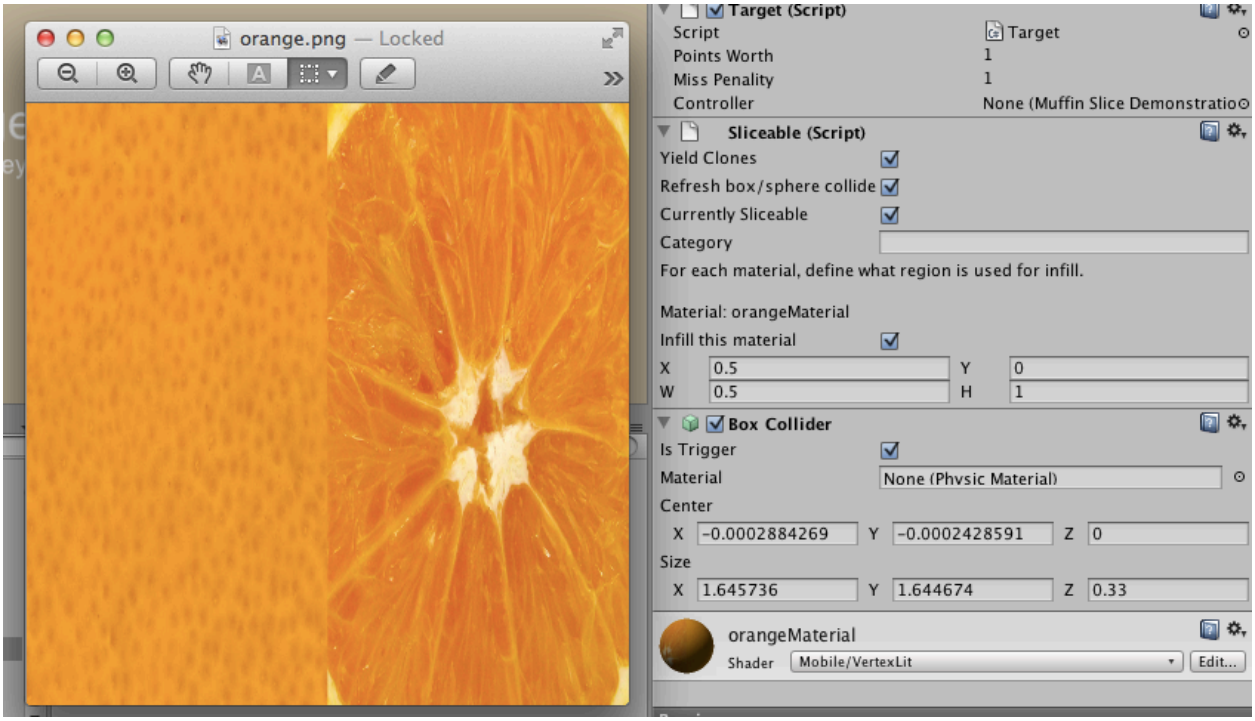


The closed surface means that if you slice it pretty much anywhere, you will get a cross section with closed polygons.

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.

Alternate Prefab

This is your character's rag-doll prefab. Limb Hacker will use this to perform a slice. **Its hierarchy must match the original character**. If you leave it blank, the character will not become a rag-doll.

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 later in this document.

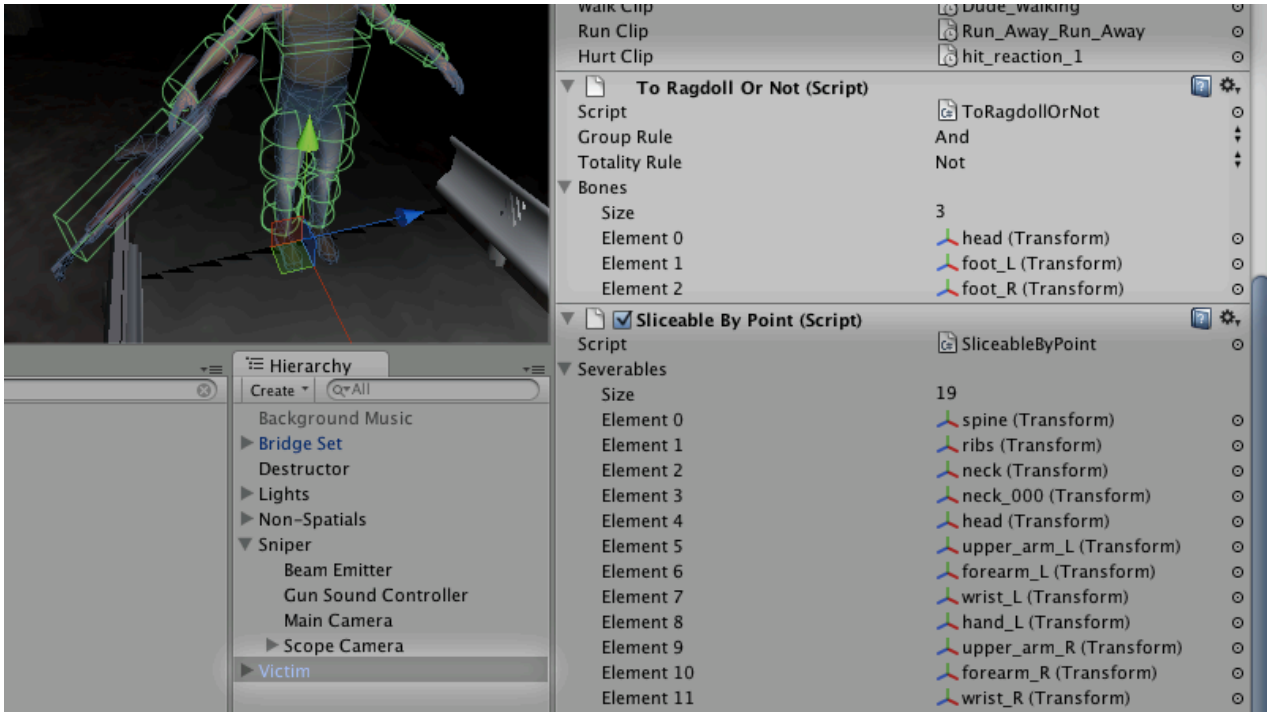
Currently Sliceable, Category

This pertains to a feature that is disabled at this time. It will be included in version 1.1.

To Ragdoll Or Not

When Limb Hacker performs a sever, it yields two objects, each containing part of the resulting geometry. It must decide whether new objects are based on the original object, or the ragdoll (or “alternate prefab”).

You can write the code for this yourself by extending AbstractSliceHandler (described below) and adding your component to the object, or you can use our pre-made component, ToRagdollOrNot.



This component is called upon during the slice. It checks for the **presence** of bones in any given slice result. In the demo, we list the head, foot_L and foot_R bones by adding their transforms to the Bones list.

In this demo, the part that remains a whole character with agency (not a ragdoll) is the one that has the head, left foot *and* right foot. (We could add more but that would be superfluous). Therefore any part that does *not* have the head, left foot *and* right foot is not a live character and needs to become a ragdoll.

If you want to copy the behavior in our demo, go ahead and copy our settings.

In detail

In boolean terms, we might say:

```
becomeRagdoll = NOT ( has(Head) AND has(foot_L) AND has(foot_R) )
```

So we set the “group rule” to “and”, and the “totality rule” to “not”. (The various items’ presence will be combined using the *and* operator and the whole will be inverted.)

Let’s see how this might play out in practice.

Example 1

Suppose we shoot off his hand. We now have two entities; one which has only the severed hand, and the other which possesses the head and both feed. For each of those entities, we evaluate the data like so:

```
becomeRagdoll = NOT ( FALSE AND FALSE AND FALSE ) = TRUE
becomeRagdoll = NOT ( TRUE AND TRUE AND TRUE ) = FALSE
```

So we see one of the resultant entities will become a ragdoll, and the other will not. In the demo, this means the hand will fall to the ground (it is a ragdoll, governed by physics) while the other result – the live character – will remain governed by its AI and drop its gun and run off.

Example 2

Suppose we sever it at the head. We now have two entities; one with the head, and the other with the rest of the body. If we drop each entities' presence data into the function, we see these:

```
becomeRagdoll = NOT ( FALSE AND TRUE AND TRUE ) = TRUE
becomeRagdoll = NOT ( TRUE AND FALSE AND FALSE ) = TRUE
```

So both pieces are ragdollified.

Whut?

If you want to mimic the behavior of the demo, go ahead and copy our settings.

If you want to write your own decider, read on.

Abstract Slice Handler

This is an abstract class that inherits from MonoBehavior. You may extend and implement its **cloneAlternate** method. (Please ignore its **handleSlice** method; this will become relevant in the next update.)

The clone alternate method permits you to decide if a given slice half will be based on the original object or the alternate prefab (usually a ragdoll).

```
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 which bones are present. You could, for example, return false if the head is not present. ToRagdollOrNot implements this method; you may examine it as an example.

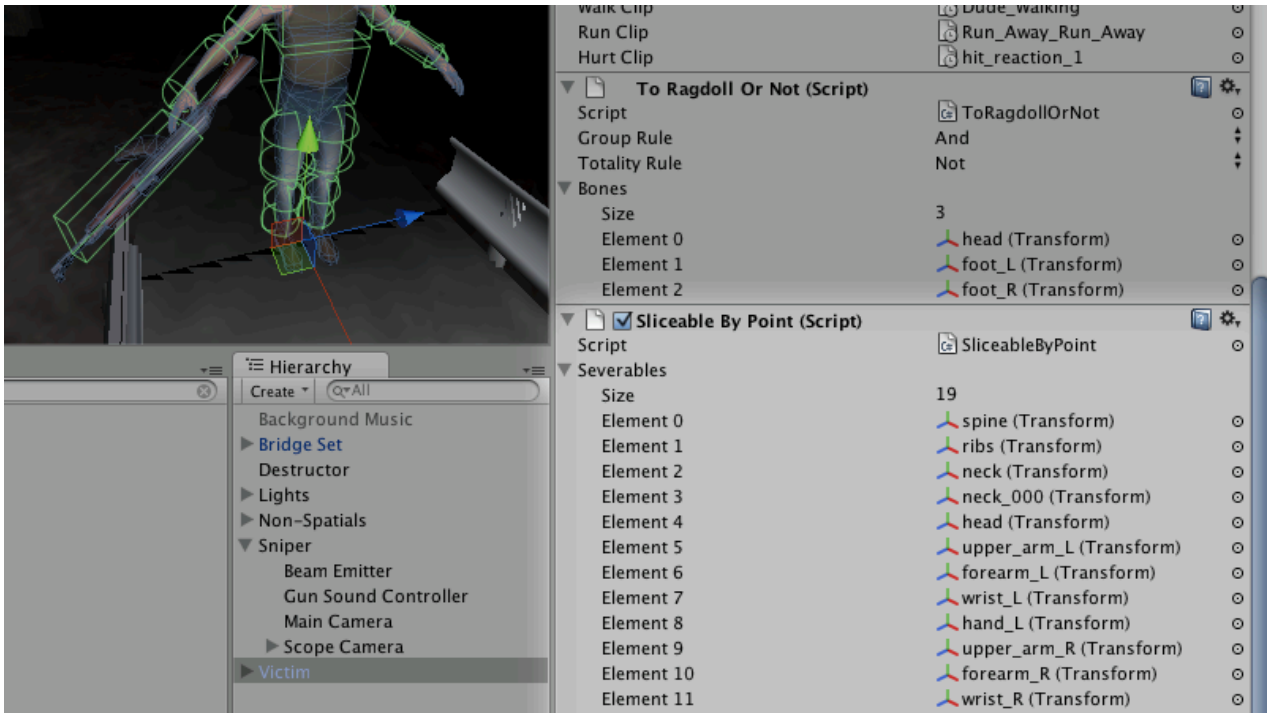
Sliceable By Point

The Limb Hacker API (described later) offers a method to slice by a given point in world space instead of specifying a joint. You might use this if – for example – you have a

point in world space from a ray cast or a collision and want to slice whatever it looks like it's supposed to slice.

However **this requires configuration**. Not all slices look good; in testing, we found that slicing (for example) the character's collar bone joint yielded ugly results.

A future release might have a sufficiently good heuristic to judge automatically if a slice is going to bork, *or* might have a more advanced infill procedure. (The latter is being investigated.) However in the meantime, this component lets you specify exactly which bones are severable.



We add this component next to the Sliceable and other components, and add the bones that are severable. In this case we added all real bones (not the AK-47 as that was not part of the skinned mesh's hierarchy) and skipped the collar bones.

With this information, the severByPoint method can avoid artifacts.

Performance

Do I need to care?

Yes, if you plan to release on mobiles. Desktops and notebooks ought to be able to handle very large data sets without noticeable lurch, however very low-end mobiles will not be able to.

General Considerations

Mesh slicing 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 – from which Limb Hacker inherits must code – 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” would be 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 the first 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.

Limb Hacker API

There are two public APIs. They are not static, but can be accessed via the static field LimbHacker.instance which will automatically create an instance if one does not exist.

```
GameObject[] severByJoint(GameObject go, string jointName, float
   rootTipProgression = 0f)
GameObject[] severByPoint(GameObject go, Vector3 reasonablyClosePoint)
```

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 **one object** (if not sliced) or **two objects** (each result of a slice).

Sever By Joint

Sever by joint will hack off part of the character from any specified joint. It has an optional parameter; *rootTipProgression*. This is a float with a range [0,1) that tells where between the specified joint and its child you want the slice to occur. (If it has multiple children, it will take their mean position.) For example if we gave it the bone name of the left elbow and a root-tip-progression of 0.5, it would slice halfway through the left forearm.

Sever By Point

Sever by point will take a position in *world space*, find the nearest bones and try to sever at the right place. For example, a position *reasonably close* to halfway between the left elbow and wrist ought to slice halfway through the left forearm.

This is used in the Sniper Demo; the script LHD2Enemy calls this method with a point taken from a ray cast hit. To get a sufficiently accurate effect, multiple colliders are placed on the live character rather than a single bounding box. This means that a ray cast can hit reasonably close to the arm its aimed at, rather than way out on the side of a bounding box or sphere.

This feature requires configuration. See “Sliceable By Point” earlier in this document.

Contact

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