Limb Hacker Guide

Limb Hacker version 1.1

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 Sniper 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.

Preparing an object

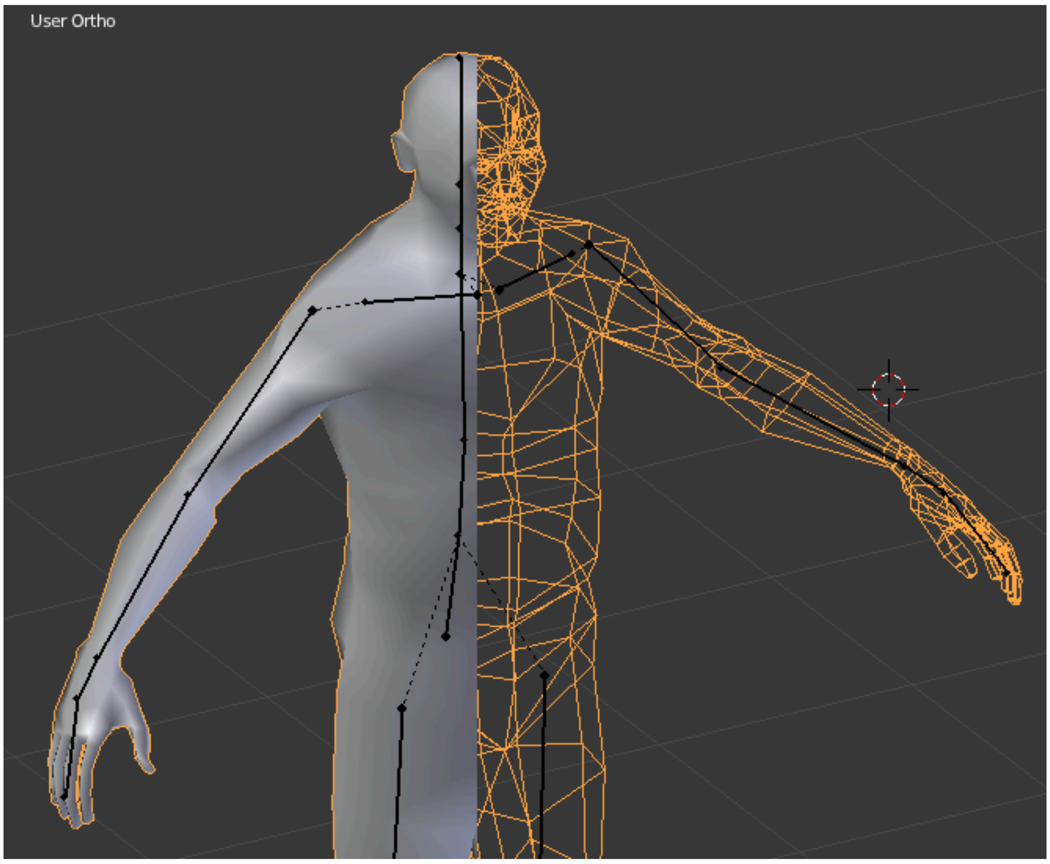
To slice an object, Limb Hacker needs to be able to find a **SkinnedMeshRenderer**. It **does** support meshes with multiple materials.

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 **Hackable** component to this object. When you feed an object to Limb Hacker, it will try to find a single Hackable component either on it or in its children and use the configuration described there.

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

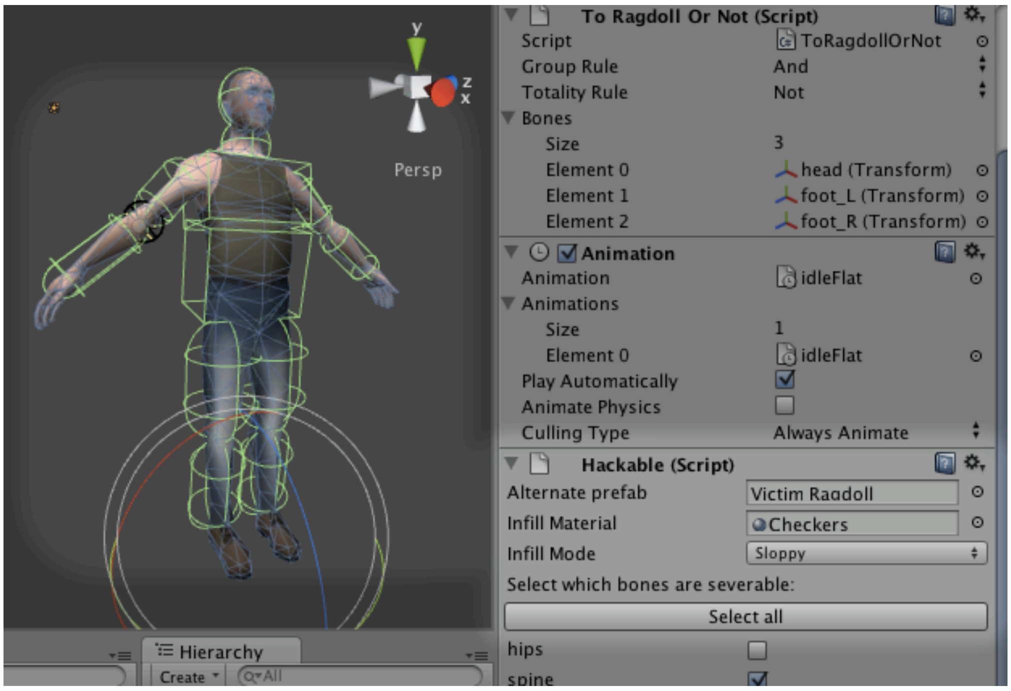
The slicer is designed assuming it will be dealing with closed, textures meshes. Meshes which have layered, hidden geometry or triangles which pass through each other may result in infill anomalies. (The "infill" is the geometry made up to cover holes made by the slice.) An ideal mesh has geometry like so:



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 any texture material provided using the Hackable component's "Infill Material" property.



To Ragdoll Or Not (Script)
Script ToRagdollOrNot
Group Rule And
Totality Rule Not
Bones
Size 3
Element 0 head (Transform)
Element 1 foot_L (Transform)
Element 2 foot_R (Transform)
✓ Animation
Animation idleFlat
Animations
Size 1
Element 0 idleFlat
Play Automatically ✓
Animate Physics ☐
Culling Type Always Animate
Hackable (Script)
Alternate prefab Victim Ragdoll
Infill Material Checkers
Infill Mode Sloppy
Select which bones are severable:
Select all
hips ☐
spine ✓

There are two infill algorithms; *sloppy* and *meticulous*.

The former is default. The name may sound less pro, but it's the one you want 99% of the time. You may drop in a texture and stop reading here, however I will go into detail in case you are encountering artifacts.

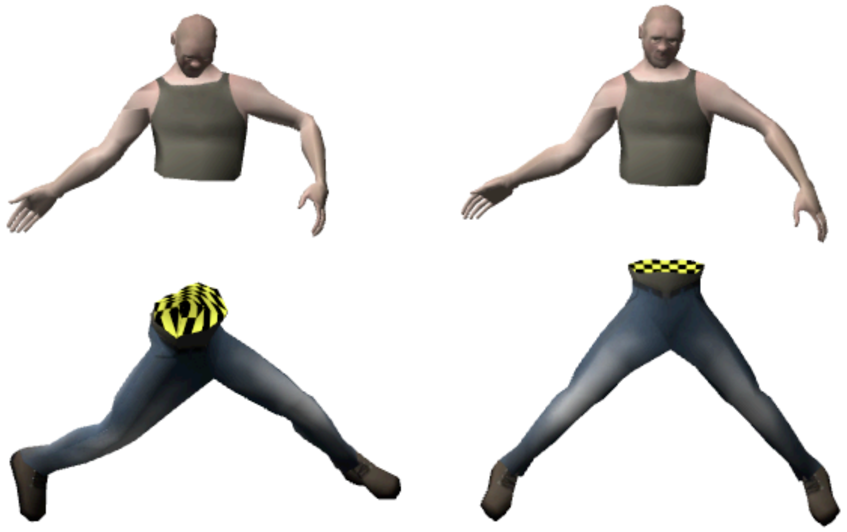
Limb Hacker is developed from a prior product – Turbo Slicer – which is designed to rapidly slice *whole* objects *repeatedly*. The italicized words are key and influence its design, including how it differs from Limb Hacker. When a whole object, like Synergy Blade's boss character, or a donut, is sliced, the cross section of the slice will often feature *multiple polygons*. If a naive or crude infill is applied, there will be hideous polygons between the slice holes. Even if there is only one polygon, it may have an irregular shape which is lends itself to artifacts. The *meticulous* infill procedure was developed for this and is the only infill algorithm in that product.

However, it has a problem that is exacerbated in Limb Hacker; if the slices itself is imperfect, it will fail to read it and abort; no infill will occur. This is exacerbated because Limb Hacker attempts to ignore irrelevant vertices – to avoid worse anomalies – and occasionally ignored vertices overlap with the slice plane. (Slicing across the shoulders is liable to do this, depending on how a mesh is skinned.)

Limb Hacker, however, is not intended to slice whole models; it is meant to slice through character limbs. The vast majority of slices either have single, regular polygons in their cross section or something close to it at a glance.

Therefore, a naive, sloppy infiller has been added to Limb Hacker. It does not attempt to distinguish polygons (a delicate process) and creates the infill using a simple triangle fan.

Here we see, using a test pattern to illustrate one difference, the texture mapping from the sloppy infill (left) next to the meticulous infill (right):



With gore textures and in motion, the difference is unnoticeable. The sloppy infiller will also perform passable infills in situations where the meticulous infill will outright refuse.

When should you use the meticulous infill? There may be some edge cases – such as the forearm of a human skeleton – where a “bone” is depicted in the mesh with multiple objects and only the meticulous infill can correctly fill each hole independently.

If slicing a particular bone looks bad with either, you can abstain from slicing that bone and configure the *sliceByPoint* feature to ignore it. How to do this is explained later in this document.

One other difference between Limb Hacker and Turbo Slicer is the requirement in Turbo Slicer that the infill texture be atlassed. This is the case there due to its goal of slicing objects to shreds from multiple angles; Unity treats each material as a “sub mesh”, which means that if what appears to be a single piece of geometry has multiple materials, it will actually be composed to separate, *open* sub meshes rather than one *closed* whole. This will break infill on subsequent slices. (This also reduces draw calls and texture count; important on lower end mobiles.) Whereas Limb Hacker does not have this design requirement, and using a single material is easier for the user, we have changed Limb Hacker to use a single material.

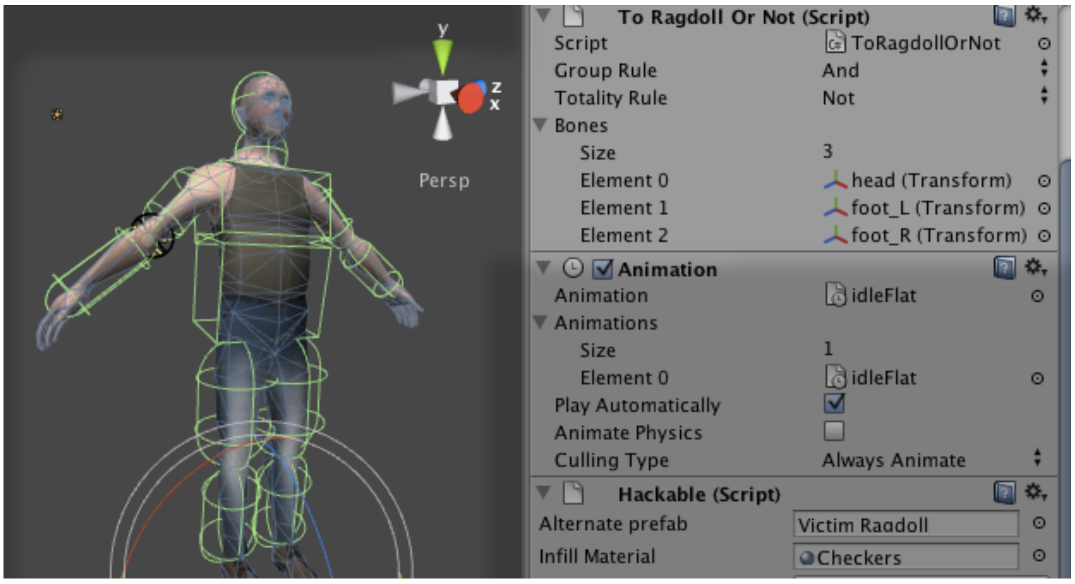
Alternate Prefab

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

To Ragdoll Or Not

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



To Ragdoll Or Not (Script)

Script	ToRagdollOrNot
Group Rule	And
Totality Rule	Not
Bones	
Size	3
Element 0	head (Transform)
Element 1	foot_L (Transform)
Element 2	foot_R (Transform)

Animation

Animation	idleFlat
Animations	
Size	1
Element 0	idleFlat
Play Automatically	✓
Animate Physics	☐
Culling Type	Always Animate

Hackable (Script)

Alternate prefab	Victim Ragdoll
Infill Material	Checkers

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 the included demos, 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).

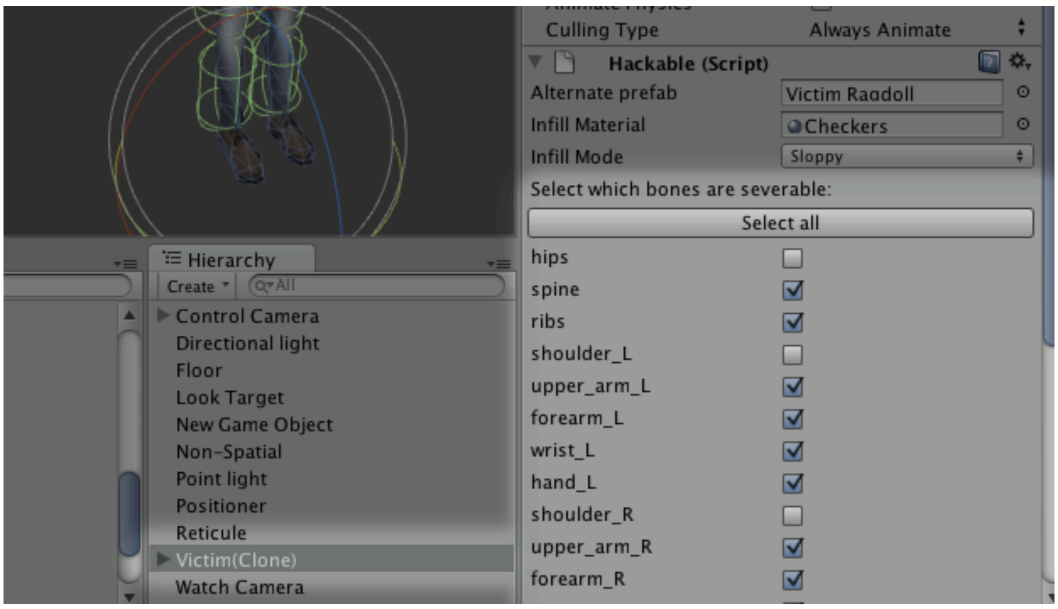
```
public virtual bool cloneAlternate ( Dictionary<string,bool>
   hierarchyPresence ) {
   bool useAlternatePrefab;
   // ...
   return useAlternatePrefab;
}
```

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.

Slice 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 sufficiently good heuristic hasn't been developed yet. In the meantime, the Hackable component lets you decide which bones are severable.



Culling Type	Always Animate
Hackable (Script)	
Alternate prefab	Victim Ragdoll
Infill Material	Checkers
Infill Mode	Sloppy

Select which bones are severable:

Select all

hips	☐
spine	✓
ribs	✓
shoulder_L	☐
upper_arm_L	✓
forearm_L	✓
wrist_L	✓
hand_L	✓
shoulder_R	☐
upper_arm_R	✓
forearm_R	✓

Only *bones* will be selectable. Objects attached to bones in the hierarchy – like the Sniper Demo's machine gun – will follow their parent 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 bone 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 “Slice By Point” earlier in this document.

Contact

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