# Limb Hacker Guide

Limb Hacker version 1.3

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 slice a nondescript figure with a mouse controlled sword.

You can slice from a particular joint with the Limb Hacker API or use the Attachable Slicer.

## 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 Unity Asset Store cover art was painted by artist Aleksandra Bartosik.

## 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.

## Preparing an object

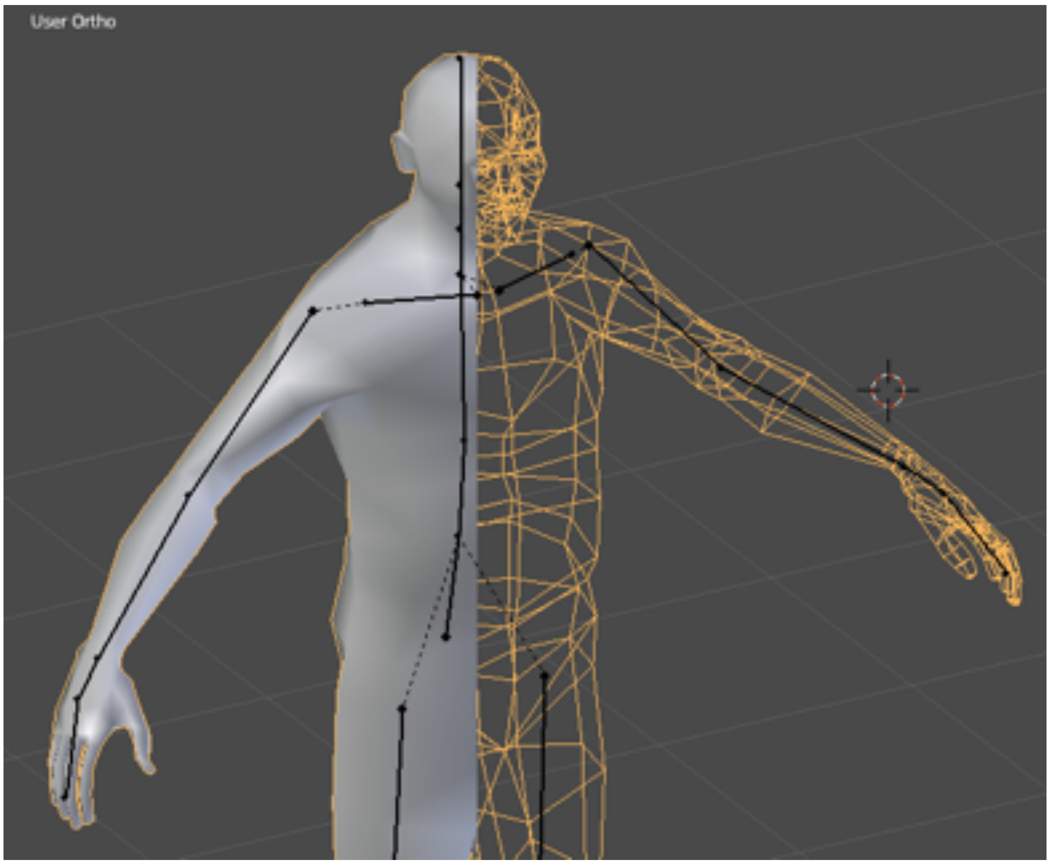
To slice an object, Limb Hacker needs to be able to find a **SkinnedMeshRenderer**. It **does** support meshes with multiple materials, but **does not** support multiple skinned mesh renderers

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 provide 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 slice handler component to the same object as the Hackable component.

### Ideal Mesh

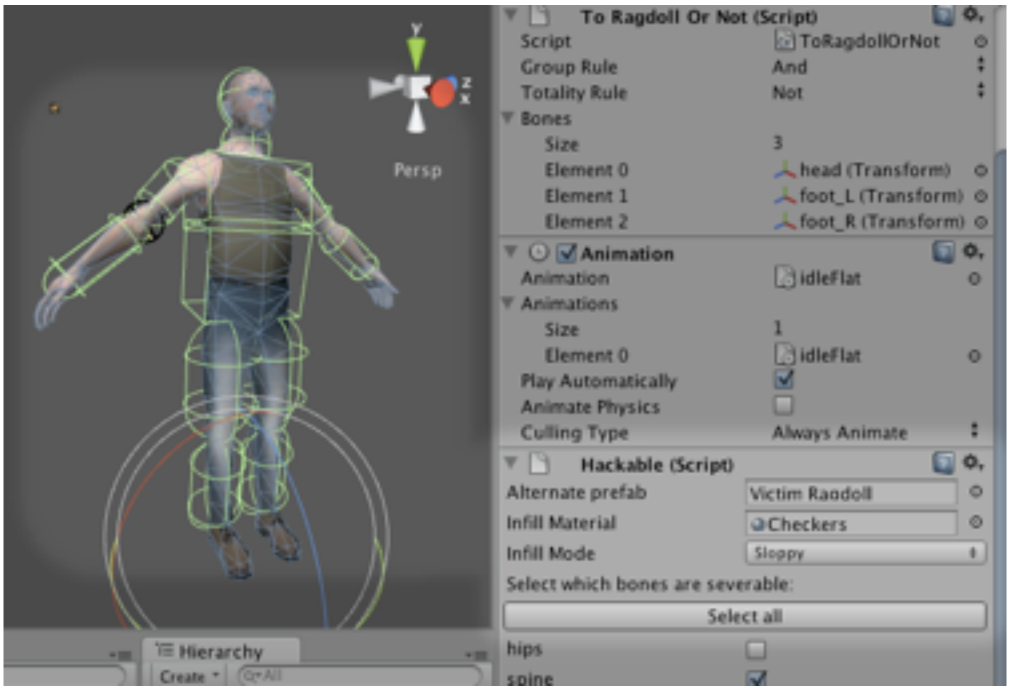
The slicer is designed assuming it will be dealing with closed, textured meshes. Meshes which have layered, hidden geometry or triangles which pass through each other may result in infill anomalies. (The “infill” is the geometry made up to cover holes made by the slice.) An ideal mesh has geometry like 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; it is both faster and more reliable. You may drop in a texture and stop reading here, however I will go into detail in case you are encountering artifacts.

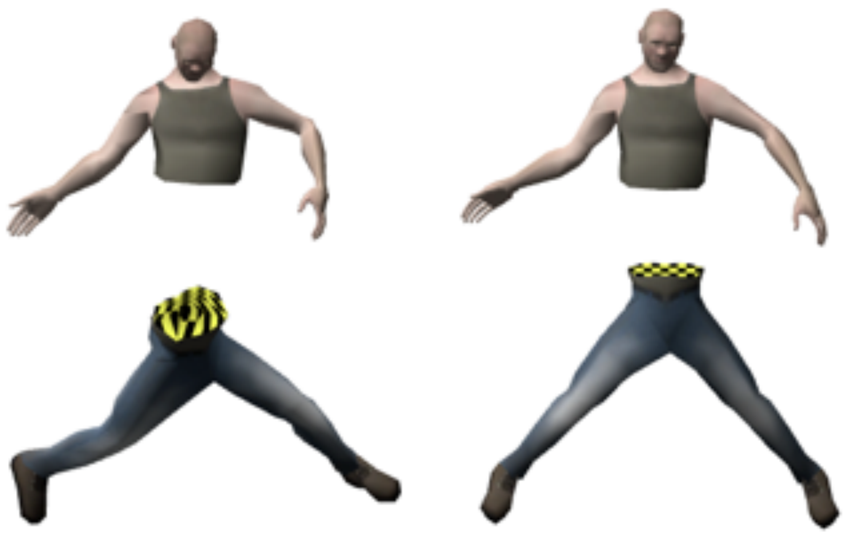
Limb Hacker is developed from a prior product – Turbo Slicer – which is designed to rapidly slice *whole* objects *repeatedly*. The italicized words are key and influence its design, including how it differs from Limb Hacker. When an object like a donut is sliced, the cross section of the slice will often feature *multiple polygons*. If a naive or crude infill is applied, there will be extraneous 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; it is able to distinguish multiple polygons in a cross section and fill them with geometry with aligned UVs. It s the only infill algorithm in Turbo Slicer.

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. Limb Hacker attempts to ignore irrelevant vertices, and occasionally ignored vertices overlap with the slice plane. (Slicing across the shoulders is likely 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 have cross sections which are single, regular polygons or at least appear as such from a distance.

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 difference in texture mapping from the sloppy infill (left) next to the meticulous infill (right):



In motion, and with organic textures, the difference is invisible. 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 by marking it unseverable. 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.

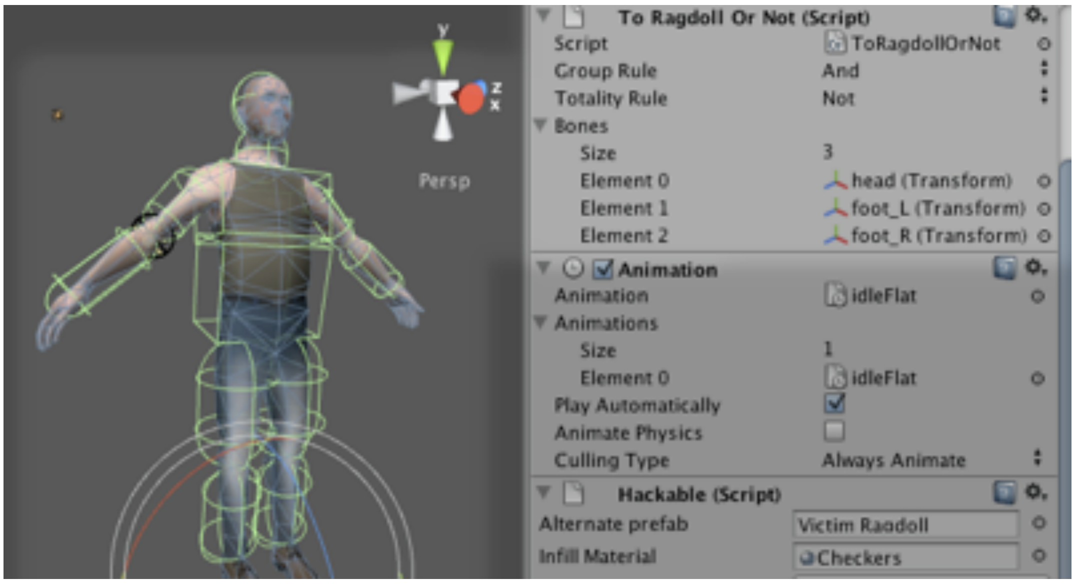
### Alternate Prefab

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 is used for Turbo Slicer.)

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).

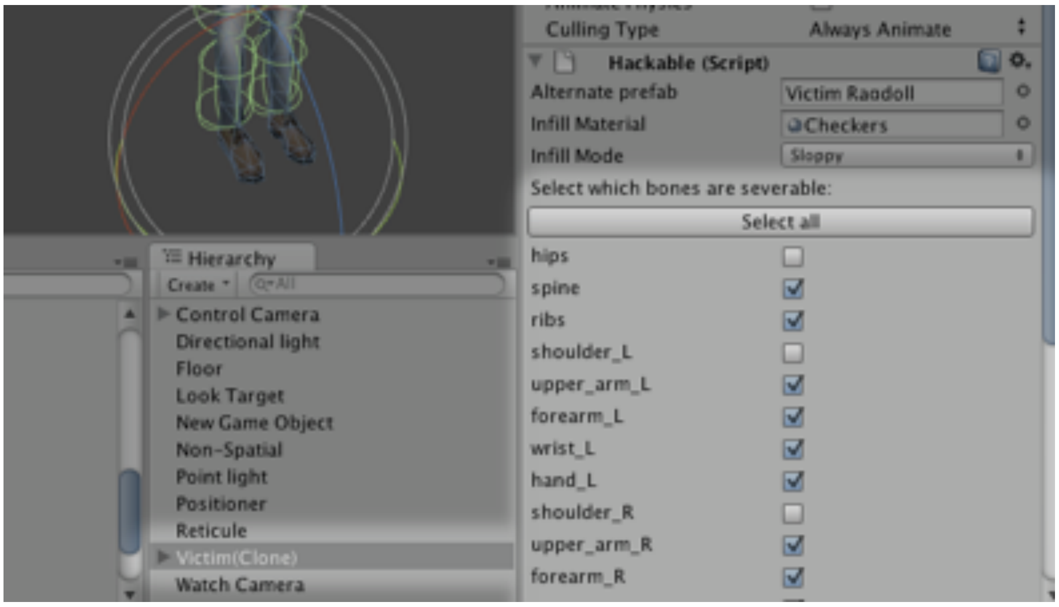
```
public virtual bool cloneAlternate ( Dictionary<string,bool>
    hierarchyPresence ) {
    bool useAlternatePrefab;
    // ...
    return useAlternatePrefab;
}
```

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 true if the head is not present, or return true if the head is the only item present, and this would cause the severing of a head to convert both resulting halves to a ragdoll. ToRagdollOrNot derives from this class; 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. The Hackable component lets you decide which bones are severable.



Control Camera
Directional light
Floor
Look Target
New Game Object
Non-Spatial
Point light
Positioner
Reticule
Victim(Clone)
Watch Camera

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 will follow their parent bones.

With this information, the severByPoint method can avoid artifacts.

# 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)
GameObject[] severByJoint(GameObject go, string jointName, float
    rootTipProgression, Vector3? planeNormal)
GameObject[] severByPoint(GameObject go, Vector3 reasonablyClosePoint)
GameObject[] severByPoint(GameObject go, Vector3 reasonablyClosePoint,
    Vector3? normalInWorldSpace)
```

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 joint specified by name.

### Root Tip Progression

Root tip progression 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.

### Plane Normal

Plane normal tilts the slice plane. The attachable slicer delivers angle information. However not all tilts are sensible. A ninety degree tilt, for example, can't slice the skeleton even if the mesh could hypothetically be sliced any way we please. Limb Hacker will attempt to restrict given plane normals to sensible tilts to avoid artifacts.

## 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.

### Normal in world space

Providing a normal in world space has the same effect as Sever By Joint's plane normal; it will control the tilt of the slice plane, within reason.

## Hackable: ISliceable

```
GameObject[] Slice(Vector3 positionInWorldSpace, Vector3
    normalInWorldSpace);
```

The Hackable component implements the ISliceable interface, which is also implemented by Turbo Slicer's Sliceable component. Because Hackable and Sliceable both implement ISliceable, code can dispatch slices to either system. The Attachable Slicer uses ISliceable.

## Contact

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