Demo Scene Quick Start

1. **Import** the Mount Point package into a new project.

2. Open the **MP_Demo scene**.
   Assets\ootii\MountPoints\Demos\Scenes\MP_Demo

3. Press **play**.
   The buttons on the left will allow you to mount weapon and armor on our hero.
Custom Scene Quick Start

1. **Import** the Mount Points package into your project.

2. Select your character and add a **Mount List** component.

3. Add **mount points**.

   Once you add mount points to this game object, you can do the same thing to another object. As you drag the objects so mount points touch, they will snap together.
Foreword

Thank you for purchasing Mount Points!

I’m an independent developer and your feedback and support really means a lot to me. Please don’t ever hesitate to contact me if you have a question, suggestion, or concern.

The latest version of the documentation can be found online:

I’m also on the forums throughout the day:
http://forum.unity3d.com/threads/235816-Mount-Points

Tim
tim@ootii.com
Overview
Unity has a pretty good system for connecting objects together. The problem is that if you have a skeleton, finding and managing those connections isn’t easy at all.

Mount Points is a super simple tool for connecting objects together and managing those relationships.

With Mount Points, you can simply drag two objects together and when their mount points get close, they snap together like magnets or Legos™. Drag the objects away from each other and they come apart. Mount Points manages the relationship, orientation, and scale for you.

Mount Points doesn’t try to recreate Unity’s hierarchy, it leverages and simplifies it.

New in version 2.0 is the ability to mount and manage skinned meshes. With the new user interface, you can set clothing up at edit-time and have them exist at run-time. Use skinned item masks to keep body parts from poking through your clothes.

Features
Mount Points supports the following features:

- Simplified object connections
- Easy access through editor or code
- Preserves child scale
- Precise position snapping
- Precise orientation snapping
- Mount points attach to bones for animating
- Supports skinned meshes
- Skinned mesh masks to hide ‘poke-throughs’
- Set clothing up at edit-time
- Supports prefabs
- Supports skinned meshes
Understanding Mount Points

Mount points are like magnets; they work in pairs. When two mount points are close together, they snap together and create a parent-child relationship.

The object being dragged is the child and the object it’s dragged to is the parent.

Colors

Mount points in the scene are colored to help determine who they belong to.

- **White** mount points simply show mount points that exist in the scene for any object.
- **Yellow** mount points are mount points that exist on the currently selected object.
- The **green** mount point represents the mount point that is currently selected in the inspector.

Unity Hierarchy

The mount point system doesn’t create its own hierarchy. Instead, it leverages what Unity has already done. That means it works natively with any other Unity system.

When a mount point is created, you’ll actually find it inside of the Unity hierarchy. In the example below, the ‘Right Hand’ mount point is a child of the skeleton’s ‘Right Hand’ bone.
By tying mount points to bones in this way, as the animation moves the bone, the mount point (and anything connected to it) moves as well.

When another object ‘snaps’ its mount point to a parent mount point, it then becomes a child in the hierarchy too.

The Mount Point Inspector will keep the Unity hierarchy in sync as mount points are created, moved, and deleted. You don’t have to manage anything yourself.
Mount Point Orientation

Once connected, child mount points orient themselves (and the GameObjects that own them) to the rotation of the parent mount point.

This allows you to take objects created from different authors or that are oriented different ways and get the same result when you mount them.

Take two swords for example. One could be created standing on its end (the right one) while another could have been created lying down (the left one). When we put them in the character’s hand, we want them to be rotated as if the character is holding them regardless of how they were created.

This is done with the mount point’s ‘orientation’ property.

When we build the mount points for the swords, we’ll make sure they are oriented to fit the parent correctly.

For the two swords, we want the mount points to be position at the hilt with the forward direction (blue arrow) along the blade.

This is because when we build the mount point for the hand, we’re going to have that mount point’s forward direction (blue arrow) coming out of the hand too.

When the sword is connected to the hand, it will automatically rotate so that its orientation matches the orientation of the hand’s mount point.
Mount List vs. Mount

To use mount points with an object, a Mount List or Mount component needs to be added to a game object. Both of these objects are containers that manage the contained mount points.

Mount List

The Mount List is a beefy container that allows you to setup multiple mount points and skinned items. Typically, this is the component you’ll use on characters, vehicles, etc.

The top section is used to manage all the mount points. Other mount points can then be connected to these.

The bottom section is used to manage the skinned mesh items that are mounted to your character. By adding items, you can instantiate them in the editor or flag them to be created at run-time.

Mount

The Mount is a lighter version that contains one mount point.

This component is handy when you’re dealing with objects that will only be children to other mount points. For example, swords, shields, tails, etc.

It should be noted that the mount point held by a Mount cannot have children themselves.
Creating a Mount Point – Example 1

Mount points must be created in the Unity editor. Once you've created mount points for an object, you can create a prefab just like you do any other object.

The mount points will be saved with the prefab.

1. Select the GameObject

2. Add the ‘Mount’ component

   The Mount Points script are part of the ‘com.ootii.Actors’ namespace and can be found under: ‘Scripts/ootii/MountPoints/Actors/MountPoints’.

3. Add a mount point

   Click the ‘+’ button under the ‘Mount Points’ section to add a mount point. Once added, rename the mount point something meaningful. It helps if mount points on the same object are uniquely named, but it’s not required.

4. Move and Rotate the Mount Point

   Using the mount point’s editor handles, we can drag the mount point to the sword’s handle (if needed).

   As discussed earlier, we know the avatar’s hand mount point will have the forward direction (blue arrow) coming out of the hand pointing in the direction of the blade. When we attach the sword to it, it will automatically rotate to have this mount point’s orientation match that of the hand’s mount point.

   So, we want to set the orientation of the mount point so that the forward direction (blue arrow) runs along the blade. Do this with the ‘Orientation’ fields in the inspector.

   ![Mount Points Editor](image)

   In this step, you’re really just going to be moving and rotating the mount points as you see fit.

5. Set Properties

   Lastly, we set the properties for the mount point.

   In this case, we want the sword to be a child and don’t expect to have anything use it as a parent. So, using a ‘Mount’ is fine.
In order to prevent the mount point from accidentally being moved or rotated, we’ll also set it as ‘Locked’ in the mount point list.

In order for a mount point to be a child and snap to a parent, it must be locked!

Summary
At this point, we’ve setup a mount point and placed it where we want it. We’ve also rotated it in the direction we want it facing. It doesn’t matter if you’re setting up a mount point you’re expecting to be a parent, a child, or both. They are all setup the same.

If needed, you could save this object as a prefab. This is great if you plan on creating the object through code later.
Creating a Mount Point – Example 2

In this example, let’s setup the mount point for the hand that would hold the sword. We won’t have as many pictures because the steps are pretty much identical.

1. Select the GameObject
2. Add the ‘Mount List’ component
3. Add a mount point
4. Move and Rotate the Mount Point

In this case, we’re going to move the mount point to the right hand. Then, we’ll rotate it (if needed) so the forward direction (blue arrow) is coming out of the hand.

5. Set Properties

Lastly, we set the properties for the mount point. In this case, we want the hand mount point to grab items (like the sword). So, we keep the ‘Allow Children’ check mark.

In order to prevent the mount point from accidentally being moved or rotated, we could set it as ‘Locked’. However, since this mount point won’t be the child of another mount point, I actually leave the ‘Locked’ flag off. The reason is that it allows me to move the mount point while it’s holding something. It’s a good way to make sure it’s in the right position.

6. Set the Bone

So, this is a new option.

If we don’t set any bone value, the mount point is relative to the GameObject’s origin. That’s fine for objects that don’t have skeletons or animations. However, if you do have animations, we actually want the mount point to move.

In this case, we want the mount point tied to the avatar’s right hand bone. So, in the Mount Point Properties, we set the bone to ‘Unity RightHand’.

With that set, the mount point will move with the bone. Anything the mount point parents (like a sword) will move as well.
Connecting Mount Points – Editor

With two or more mount points setup, it’s now just a matter of moving the GameObjects so that the mount points are close enough to attract.

Remember that the object you’re moving is the child. When you let go of the drag handle, the child object will move to the position of the parent mount point (assuming they are close enough). Following our examples above, you’d move the sword so it’s mount point is close to the right hand’s.

What you’ll see is that the child’s inspector will change a bit. The new parent object + mount point names will display along with an options to select the parent object or break their connection.

If you were to look at the parent mount point, you’d see that it lists the children connected to it:
**Code**

Connecting mount points through code is just as easy.

There are several functions on the MountPoints and Mount objects to help you out.

Note: The “Mount List” component is actually the “MountPoints” class defined in the MountPoints.cs file.

Following standard Unity coding practices, you can get the MousePoints and Mount component from the GameObject using the following syntax:

```csharp
MountPoints mMountPoints = rParentObject.GetComponent<MountPoints>();
Mount lMount = rParentObject.GetComponent<Mount>();
```

Once you have the MountPoints (or Mount) object, you can use the following functions:

- CreateMountPoint
- ConnectMountPoints
- DisconnectMountPoints
- GetMountPoint
- GetSkinnedItem
- AddSkinnedItem
- RemoveSkinnedItem

**Code Sample**

In the demo, I use a C# file named SampleUI.cs. In it, you’ll find several ways in which I connect objects through code. For ease, I’m listing key parts here so you can see how I’m using MP in code:

```csharp
mMountPoints.ConnectMountPoints("Right Hand", GameObject.Find("Sword"), "Mount Point");
mMountPoints.DisconnectMountPoints(lSwordMP);
mMountPoints.ConnectMountPoints("Left Arm", GameObject.Find("Shield"), "Mount Point");
mMountPoints.ConnectMountPoints("Head", "Prefabs/Armor/Helmets/Helmet", "Head");
mMountPoints.RemoveSkinnedItem("Hair");
mMountPoints.AddSkinnedItem("Prefabs/Armor/Shirts/Shirt_02", "Prefabs/Armor/Shirts/Shirt_02_mask");
mMountPoints.RemoveSkinnedItemFromPath("Prefabs/Armor/Shirts/Shirt_02");
mMountPoints.AddSkinnedItem("Prefabs/Armor/Pants/Pants_02", "Prefabs/Armor/Pants/Pants_02_mask");
mMountPoints.AddSkinnedItem("Prefabs/Armor/Shoes/Boots_02", "Prefabs/Armor/Shoes/Boots_02_mask");
```
Content

In order to instantiate skinned items at run-time using code, prefabs and mask textures need to be placed in a ‘Resources’ sub folder. This will tell Unity to package the assets up even if they aren’t currently active in the scene.
Skinned Meshes

In additions to managing static meshes through mount points, this asset can also manage skinned meshes in order to clothe your character, add appendages, etc.

One of the advanced features of Mount Points is the ability to apply a ‘Body Skin Mask’ with each skinned mesh. This greatly reduces penetration of the underlying body through the mesh.

Using body skin mask, you can create “skin tight” clothing without worrying about the character popping through.

Note: Some solutions use bone manipulations and blend shapes to morph their characters. Most bone manipulations will carry through to the skinned meshes. However, blend shapes do not as those control vertices and not the bones.

Body Skin Renderer

When Mount Points ties the clothing to the body, it needs to look at the bones in the body as well as grab the material being used.

You’ll need to set the SkinnedMeshRenderer used by your character in the Body Skin Renderer field.

If you leave the field empty, Mount Points will try to find the right SkinnedMeshRenderer, but if you have multiple ones... there’s no guarantee it will pick the right one.
Skinned Items

Skinned items can be created at edit-time or run-time.

To create items at edit-time, simply enter the resource path to the mesh and a resource path to the mask.

When no instance exists, press the blue “+” icon in the skinned item details to create it or allow it to be created when the game starts using the ‘Instantiate On Start’ check box.

Unity Paths

In order to create objects at run-time, Unity needs your assets in a “Resources” folder. Otherwise, if the asset isn’t referenced in the scene... it won’t be part of your build and the path will fail.

To learn more about Unity’s “Resources” folders, you can read about them here: http://wiki.unity3d.com/index.php/Special_Folder_Names_in_your_Assets_Folder

Skinned Meshes

Skinned meshes are tied to the bones they wrap. That means you can’t simply take a skinned mesh built for one skeleton and use it for another. Since the skin is tied to each bone’s position and rotation, the skin would animate oddly if it were used on a different skeleton.

Ensure your skinned meshes are built for the skeleton you’re wrapping.

To learn more about creating skinned meshes, see the “CustomContent” document found in the Assets/ootii/MountPoints/Extras folder.
Skinned Item Mask

A Skinned Item Mask is a black and white image based on the full body texture of the character you’re attaching the skinned mesh to. When running, the masks will be applied to the diffuse texture in order to turn parts of the character invisible. White areas of the mask allow the diffuse texture to be seen. Black areas turn the corresponding parts of the diffuse texture invisible.

Once you hit play, the mask above will cause the characters legs to become invisible and the skin won’t bleed through the pants.

To use this feature, your skin’s material’s shader needs to support transparencies. I typically set the Rendering Mode to “Cutout”. For some characters like Morph3D and Mixamo, it means changing the shader your skin material uses.

Morph3D Suggestion
Assets/MORPH3D/Content/StarterPacks/Male/MCSMale/M3DMaterials/Genesis2Male.mat

Mixamo Suggestion
Assets/MORPH3D/Content/StarterPacks/Male/MCSMale/M3DMaterials/Genesis2Male.mat
Support
If you have any comments, questions, or issues, please don’t hesitate to email me at support@ootii.com. I’ll help any way I can.

Thanks!

Tim