Support

Contact

More Tools

Thank you for using Gore Simulator.

By owning this tool, you'll automatically receive a 50% discount on:

- Property Engine - A solution designed to massively improve your productivity.
- Spawn Machine - The ultimate spawning solution for Unity.

SUMMARY

INSTALLATION

Requirements

- Minimum Unity Version 2021.2 +

Dependencies

- unity.burst
- unity.collections

Installation

If you have other tools from Pampel Games installed in your project, make sure to update them to the latest version before installing Gore Simulator. This will ensure that all tools work together seamlessly.

To install Gore Simulator, simply use Unity's Package Manager like you would with any other Unity package.

The 'PampelGames' folder can be moved to a different location within the 'Assets/' folder if desired.

Shader Update

The included shaders are designed to function seamlessly with all Unity render pipelines right from the start. If you happen to encounter any visual issues, it is recommended to update the shaders manually. To do so, just follow these steps:

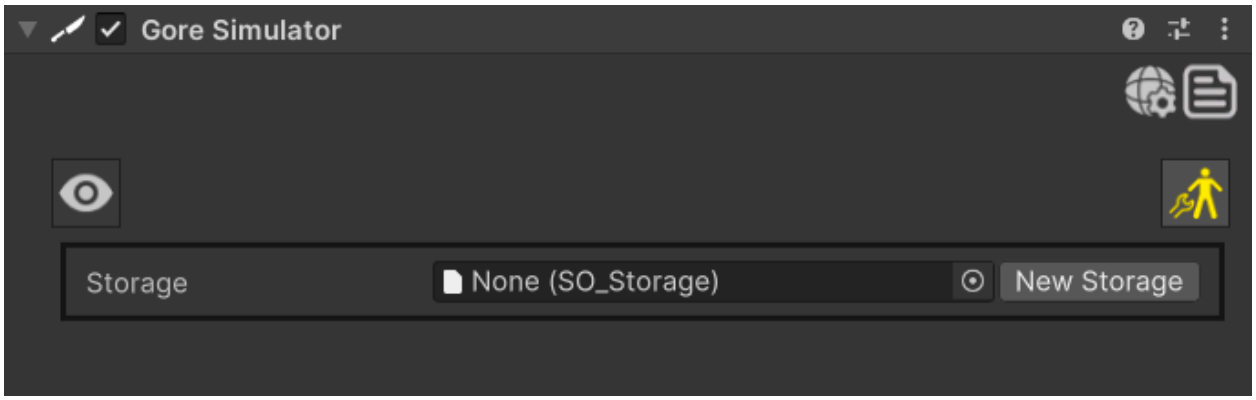
- Navigate to the PampelGames/GoreSimulator/Content/Shaders folder.
- Open each of the shader graph files (the ones with the blue icon).
- For each file, click on "Save Asset".

And that's it!

QUICK START

Initialization

To get started, simply add a Gore Simulator component to any GameObject. That being said, Gore Simulator requires a Skinned Mesh Renderer, so it would make sense to directly parent it to your characters.

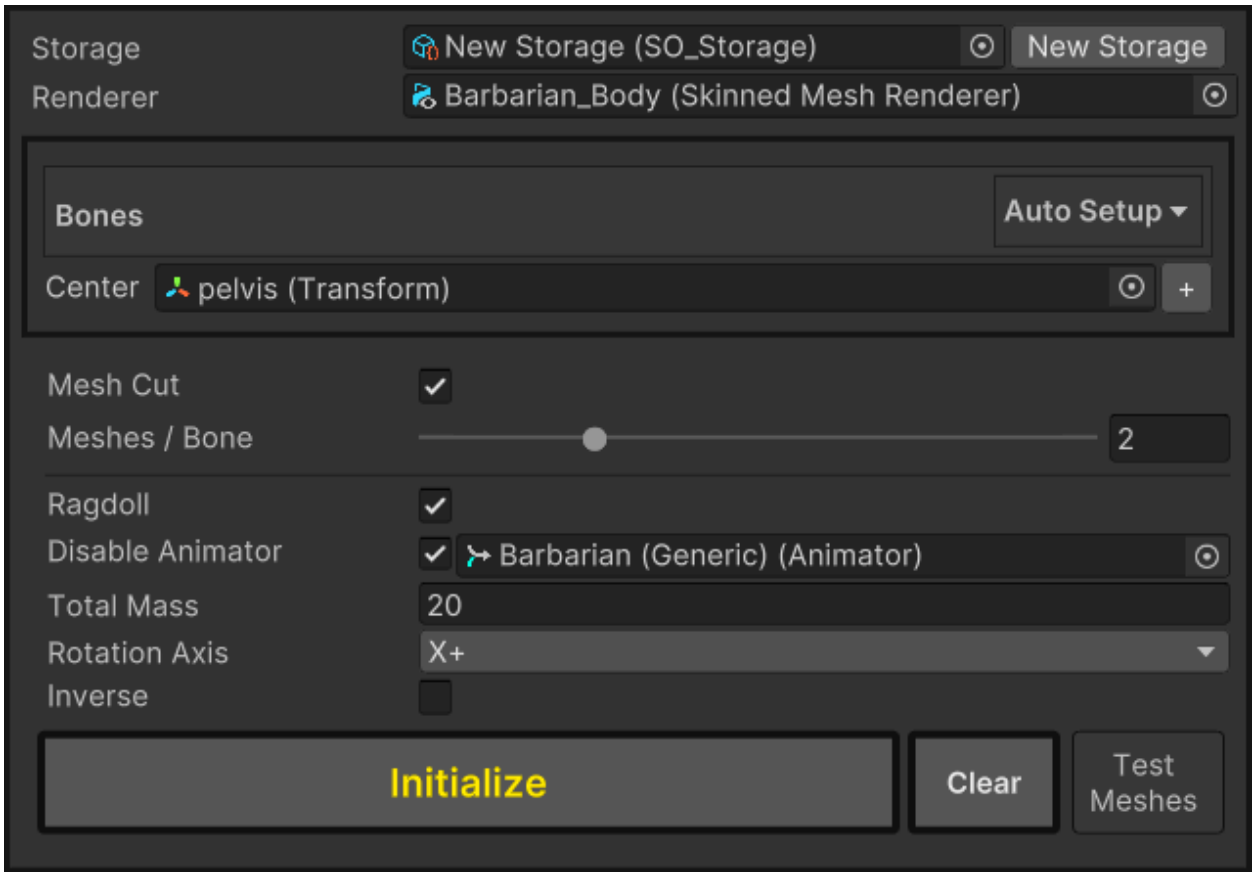


Each unique mesh/bone setup requires a storage file. Create one by using the New Storage button.

A new field becomes visible. Assign the Skinned Mesh Renderer component.

If you have multiple Skinned Mesh Renderers you want to include, please see here.

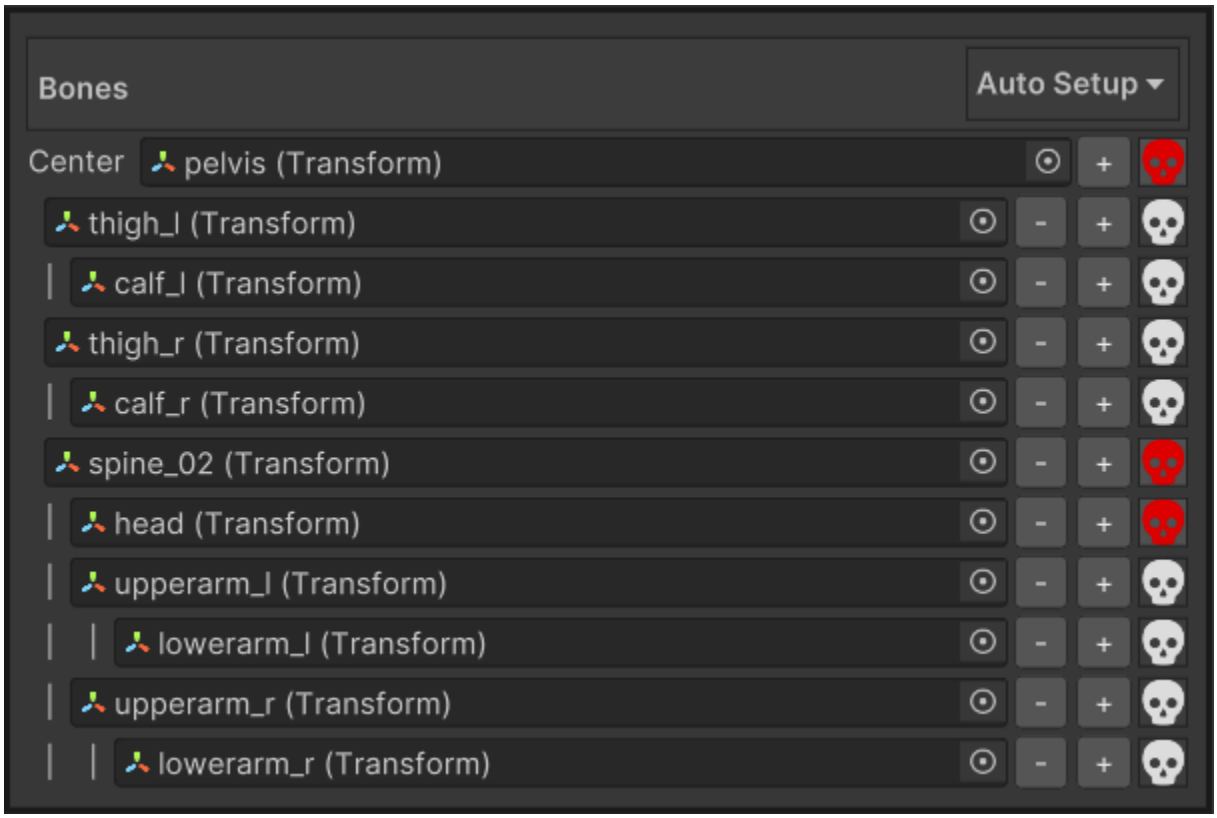
A new set of options becomes visible. In general, all buttons and fields that are not self-explanatory have detailed tooltips explaining their functionalities.



Now, a bone hierarchy has to be created. A good starting point would be to simply use one of the automated setups from the "Auto Setup" dropdown. After the hierarchy has been created, you can add or remove bones as you wish.

As a rule of thumb, use as few bones as you can but as many as needed. More details for a proper bone setup will be provided later. Just ensure that the bones are within the correct hierarchy, resembling the actual structure of the character.

Example hierarchy:



There are two main features that can be initialized for the character: Mesh Cut (+Explosion) and Ragdoll. For now, leave everything checked and click the "Initialize" button.

If the setup window does not disappear, check for any errors in the console and retry after resolving potential issues.

If you're dissatisfied with the result, be sure to press the 'Clear' button before you re-initialize.

If the initialization was successful, the component should appear something like this:

The tool has a fully modular design. The "main" modules are Cut, Explosion and Ragdoll, and each of those has sub-modules.

You can select from various sub-modules by clicking the dropdown menus on the right. If you are unsure about what a module does, you can simply add one and then hover over it (or its variables) to get more information. For the moment, leave the sub-modules empty.

In your character hierarchy, "GoreBone" scripts have been added to the assigned bones, along with colliders, rigidbodies and character joint scripts.
The automated setup is not perfect, and in most cases, especially for generic characters, some manual adjustments are needed for the best results. But more on this later.

Mesh Cut & Explosion

Now, we just need to do a little bit of coding to execute the modules at runtime. You can find a very simple example script in the PampelGames/GoreSimulator/Demo/Scripts folder. Create a new GameObject in your scene and add the "DemoExecuteMesh" script to it.

If you take a quick look into the script, you can see the recommended approach for executing Gore Simulators by using the "IGoreObject" interface, as this also covers multi-cut operations.

```
if (!hit.collider.transform.TryGetComponent<IGoreObject>(out var
goreObject)) return;

goreObject.ExecuteCut(hit.point);
```

Lastly, ensure that the game camera is focused on the character, and hit play.

As defined in the script, when clicking the right mouse button on the character, an explosion will be triggered.

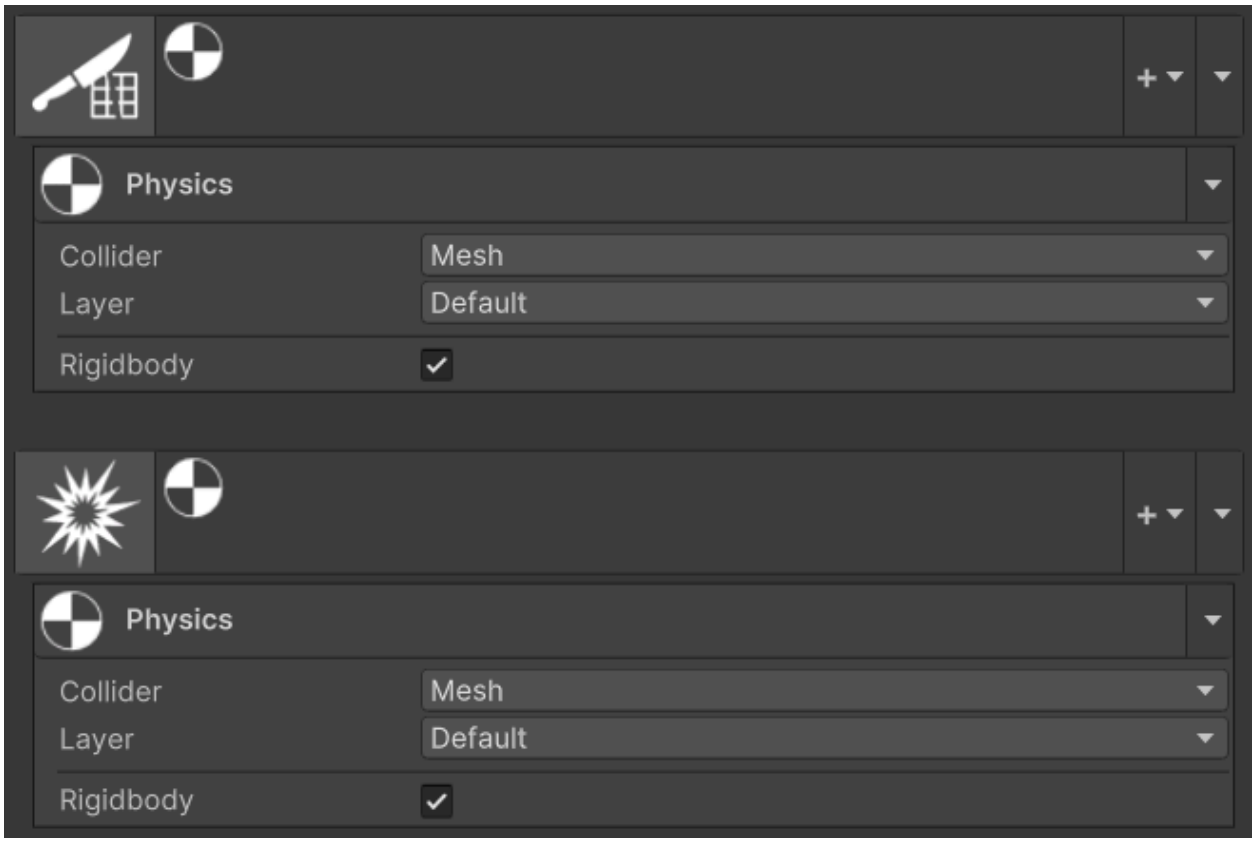
You may be wondering why nothing appeared to happen, but actually, something did occur. While still in play mode, click on the character in the scene view. You can see that the character is now composed of multiple mesh parts, which are not being animated. These are the parts resulting from the explosion.

If you hit the "Space" key, the character will be reset, as specified in the demo script.

Stop play-mode and go back to the Gore Simulator component.

Add a "Physics" sub-module to both the Cut and the Explosion modules via the "+" dropdowns on the right.

Choose a collider and also toggle Rigidbody.



Let's hit play again.

Trigger some cuts, multi-cuts, explosions, and resets as you wish. The parts are now visibly moving as they have colliders and rigid bodies attached to them.

If you like, also try out the other sub-modules. All included materials and particle effects can be found in the GoreSimulator/Content folders.

Ragdoll

Lastly, to the ragdoll module. Usually, a proper ragdoll needs some fine-tuning, but for now, let's just quickly try out the execution.

Add the “DemoExecuteRagdoll” script to your new GameObject.

To execute a ragdoll (or reset), you will need to have a reference to the Gore Simulator component. In this script, we quickly search for it in the scene and then execute it.

```
if (Input.GetKeyDown(RagdollKey))
{
     FindObjectOfType<GoreSimulator>().ExecuteRagdoll();
}

if (Input.GetKeyDown(ResetKey))
{
     FindObjectOfType<GoreSimulator>().ResetCharacter();
}
```

Note that the Ragdoll module has the "Disable Components" submodule added to it by default. This is because a ragdoll won't function if an Animator is interfering with the physics engine. The same applies to most custom character controllers, which should be disabled during the ragdoll simulation.

Congratulations on completing the quick setup!

CHARACTER SETUP

Mesh Cut

Meshes, or more specifically mesh data for specific bones, are pre-cached during initialization in the editor. While this provides significant performance benefits at runtime, it comes with the limitation that the mesh cannot be cut anywhere in any direction.

Additionally, the GameObjects used for cut and explosion operations are automatically pooled for efficient resource management. However, it's important to note that there is still some unavoidable overhead during execution.

The Meshes/Bone slider in the setup window enables you to specify the number of pre-cached meshes per bone.

The amount for specific bones could still be reduced if the algorithm identifies issues with certain parts, such as having too few vertices or overlapping vertices with a parent.

You can verify the mesh parts while in the editor by enabling the 'Test Meshes' toggle before initialization. This will instantiate the exact same mesh parts into the scene that would be used at runtime.

Multiple Skinned Meshes

A single Gore Simulator can manage an infinite number of sub-meshes within the assigned Skinned Mesh Renderer. However, when dealing with multiple Skinned Mesh Renderers for a single character, some initial steps are required. Here are some solutions:

LOD

For Skinned Mesh Renderers used only as LOD, those have to be deactivated during execution.
Tip: You can use the “Disable Components” sub-module.

Sub-Meshes

- If you intend to cut the mesh, it is necessary to merge the meshes. Fortunately, a dedicated tool is provided for this purpose. It can be accessed either through the icon located at the top right in the inspector or through the menu bar:
  Tools > Pampel Games > Gore Simulator > Combine Skinned Meshes

- If you do not want to cut the mesh, you can add the Skinned Mesh Renderers to the Skinned Children list (displayed via the "Children Setup" icon at the top). These meshes will not be cut, but a baked mesh will be created for them on execution.

Ragdoll

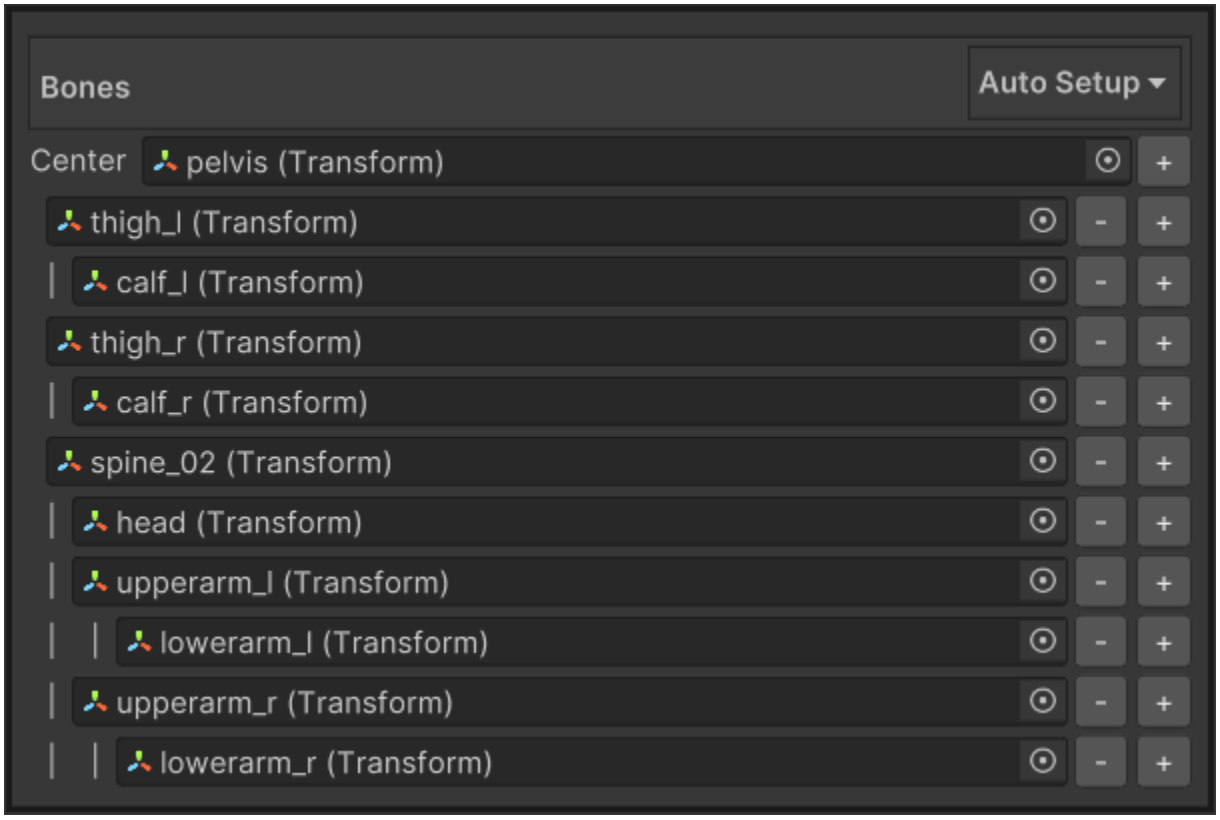
Setting up a proper ragdoll typically involves some manual adjustments.

To begin, you can use the automated setup as a solid starting point and make further adjustments from there.

Generally speaking: Use as few bones as possible, but ensure you have enough to handle all main-joints. Avoid assigning more than one bone to each main-joint; for instance, adding both the upper arm and clavicle may yield suboptimal results. Additionally, try to avoid bones with heavy interconnections, such as the clavicle.

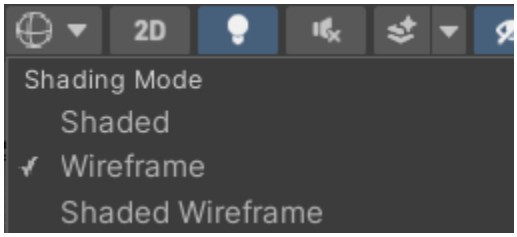
Lastly, ensure the hierarchy order aligns with the character's actual structure.

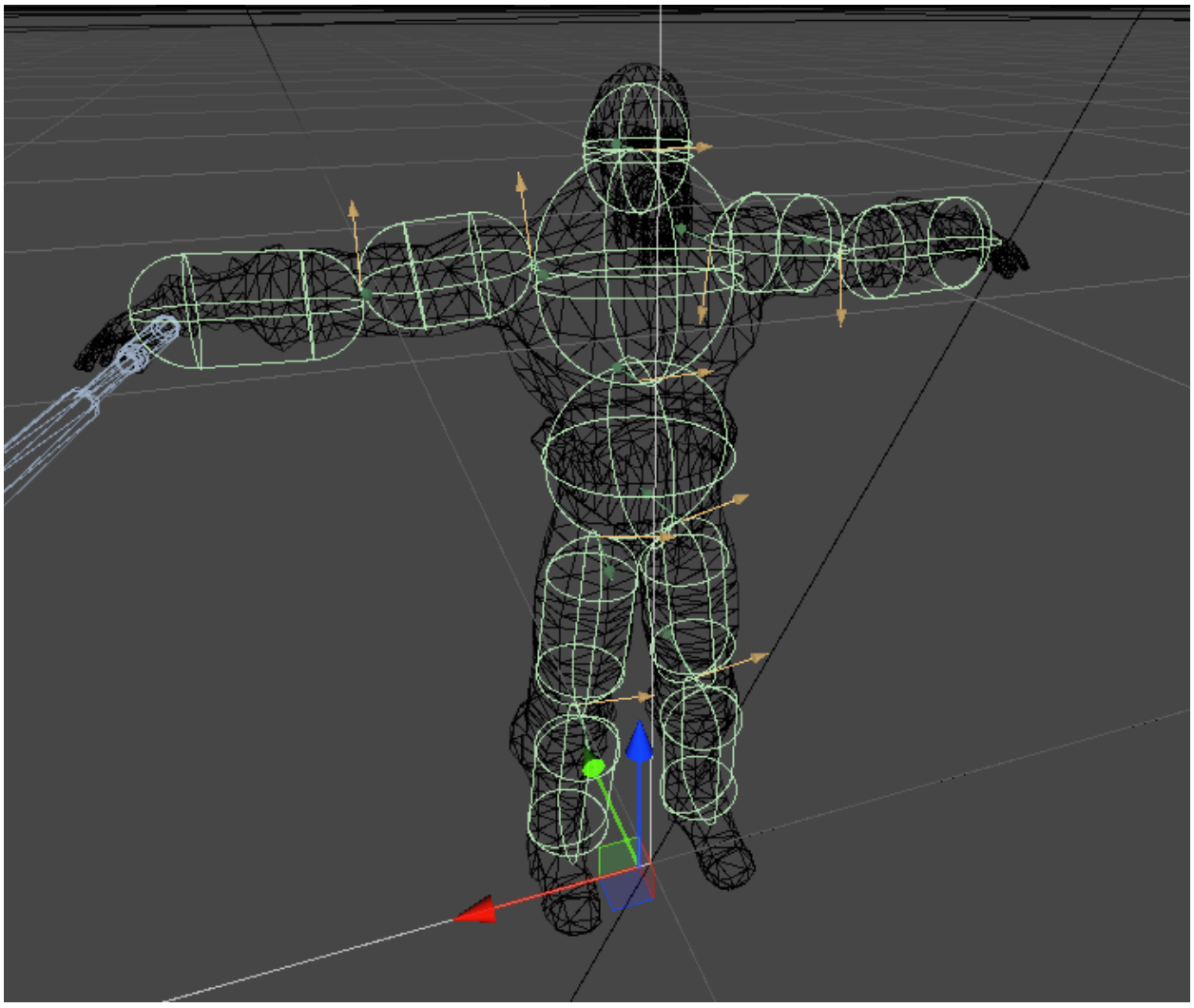
A common and correct bone assignment for humanoids would look something like this:



Note that smaller bones, such as the hands, feet, and neck, but also spine_01 and spine_03, have been omitted. Depending on your requirements, you can add them as needed.

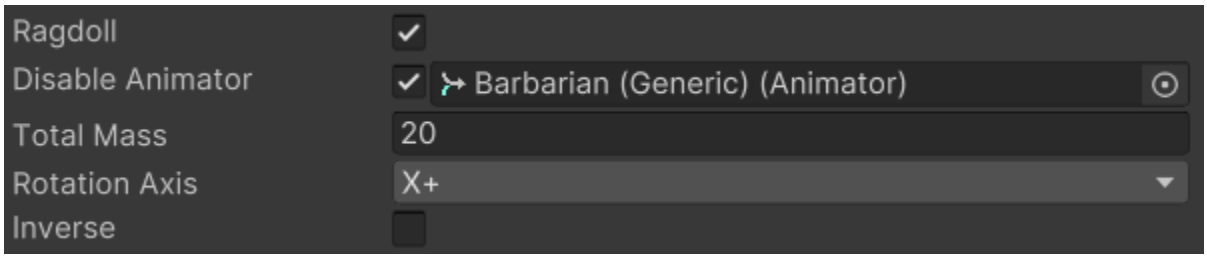
After initialization, you can visualize the character joints effectively by enabling the 'Wireframe' shading mode in the scene view and clicking on the root joint of the character.





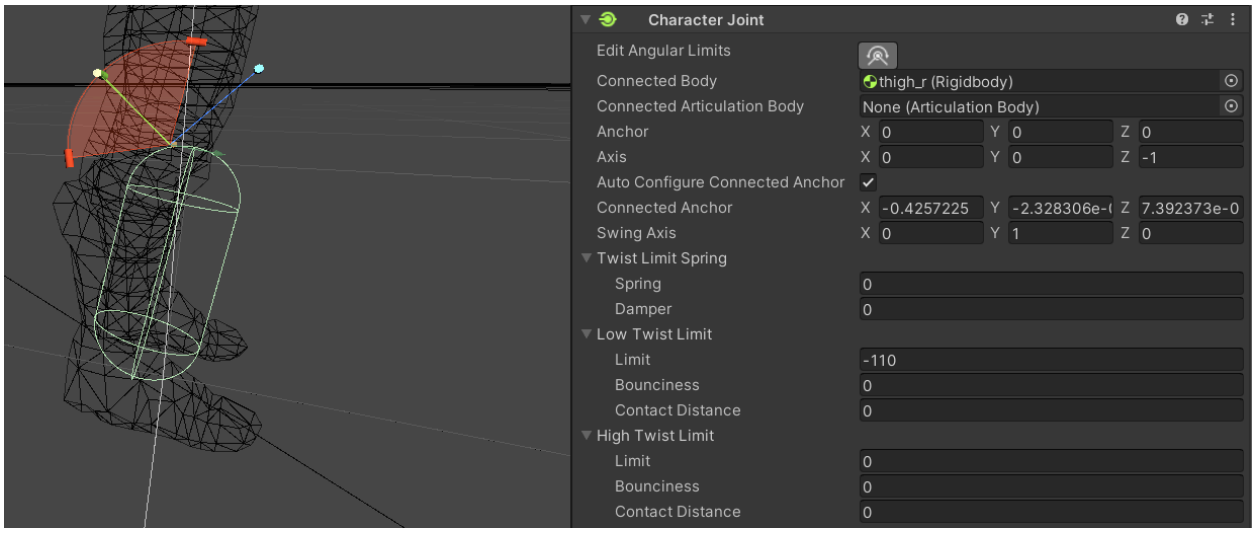
As shown in the image, Character Joint component arrows should represent the direction of joint rotation, not the direction of the joint.

If the arrows are inaccurate, for example indicating the joint direction instead, a quick fix is to clear the components using the 'Clear' button. Then, experiment with different axis orientations in the setup window.



It's also possible that some joints are correct while others are not, or that the resulting twists are inverted only for specific bones.

Navigate through your character's bone hierarchy, locate the Character Joint components, and click the 'Edit Angular Limits' buttons to visualize the twists in the scene.



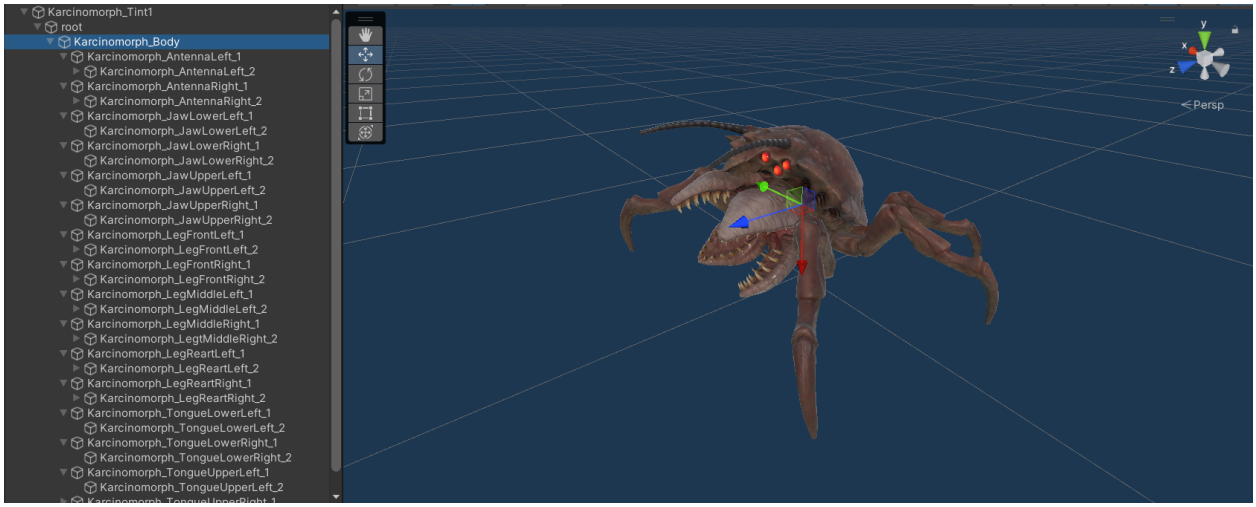
Adjust the settings for each joint accordingly and use the ExecuteRagdoll() method on the character to try them out until you are satisfied with the results.

Quick tip: If a joint, especially the head, won't stop moving, try increasing the collider size for that joint but make sure colliders are not overlapping either.

Generic RIG

Gore Simulator can be used with both Humanoid and Generic skeletons. Although they function similarly, there is a distinction in that the generic ones may require a bit more manual preparation.

Here's an example using a rather complex model with multiple legs, jaws, antennas and a deep hierarchy:



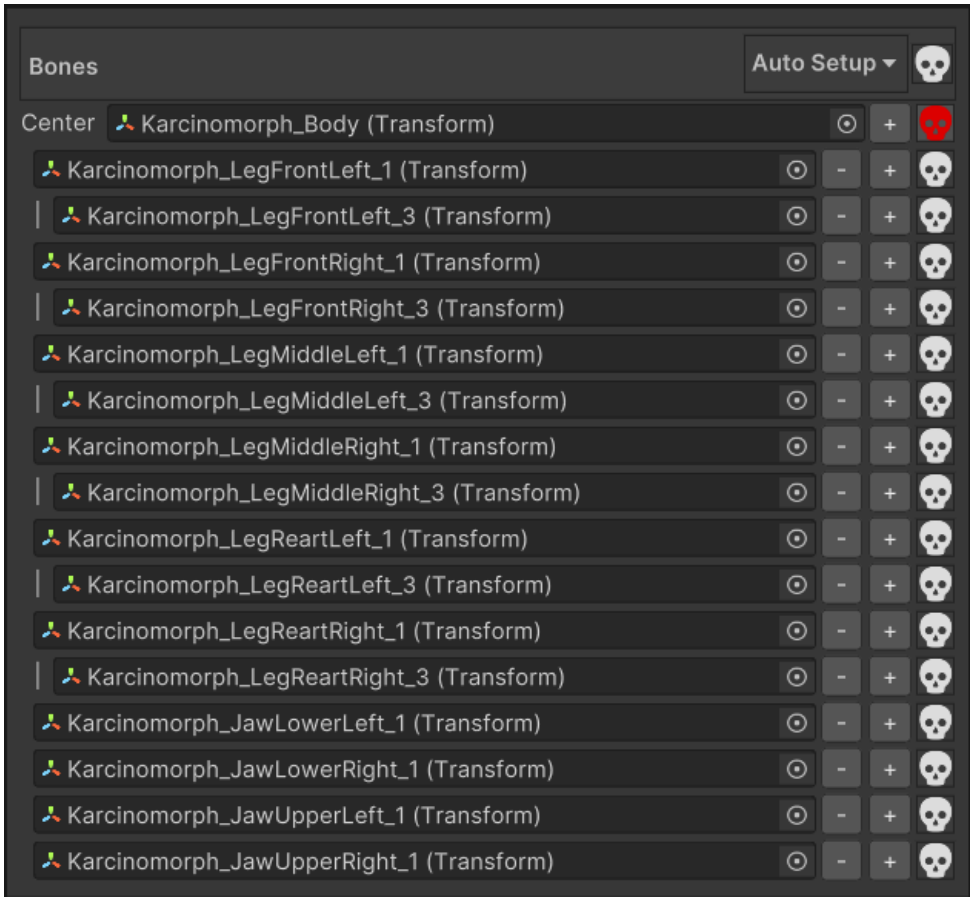
Generic character model.

Part of the nested bones hierarchy.

The auto-setup will not work properly for this model, so the bones have to be assigned manually.

The central bone is the uppermost bone in the middle of the character; the other bones can be assigned according to the actual hierarchy.

As with humanoids, you want to use as few bones as possible and only include those that are necessary. For example, legs usually only require two joints for a ragdoll. Other bones, such as the antennas, have been entirely omitted. Keep in mind that the more bones you assign, the more setup is required for the ragdoll, and the higher the CPU costs associated with it.



Generic RIG bones setup example.

MODULES

General

Gore Simulator was designed with a modular architecture. The main modules—Mesh Cut, Mesh Explosion, and Ragdoll—can only be added and removed during initialization.

The main modules are executed using the methods displayed in the tooltip when hovering over one of the modules. For example:

```
GoreSimulator.ExecuteExplosion()
```

Each main module has a set of sub-modules that can be added through the '+' dropdown on the right side of each module and removed from the dropdown on the right side of the sub-module itself, which are displayed by clicking on the big icon of a main module.

Sub-modules are being executed automatically with the execution of a module.

Decals

Decals can be added to both the Cut and Explosion modules via the ‘Decal’ sub-module. The material used for decals can be specified via the ‘Material References’ icon at the top.

Feel free to adjust the settings or apply custom textures to the material.

Using a different shader is generally not necessary and is not recommended, as the default one is optimized to seamlessly integrate with the tool. However, if you insist on using a different shader, please ensure that you include a Texture2D property named as specified in the ‘ShaderConstants.MaskTexture’ field.

Custom Modules

If you are using assembly definitions, either create your custom modules directly within the GoreSimulator/Scripts folder or reference the PG.GoreSimulator assembly definition in yours.

To create a custom module, simply derive from the SubModuleBase class. This allows you to add the sub-module to the main modules in the inspector, and it will be executed accordingly.

```
using System.Collections.Generic;
using PampelGames.GoreSimulator;

public class MyCustomSubModule : SubModuleBase
{

    public override string ModuleName()
    {
        return "My custom module";
    }

    public override void ExecuteModuleCut(SubModuleClass
subModuleClass)
    {

    }

    public override void ExecuteModuleExplosion(SubModuleClass
subModuleClass)
    {

    }

    public override void ExecuteModuleRagdoll(List<GoreBone>
goreBones)
    {

    }

    public override bool CompatibleExplosion()
    {
        return false;
    }
}
```

INTEGRATIONS

General

Gore Simulator seamlessly integrates with practically any custom character controller without requiring additional integrations.

Depending on your specific requirements, you may only need to disable the abilities associated with the affected bones or disable the entire controller once you execute ragdoll or explosion operations.

This process can be facilitated with the included Unity Actions and Events, along with the character Reset() method, if you plan to pool or reuse your characters.

If you need further assistance integrating your character controller, please don’t hesitate to reach out.

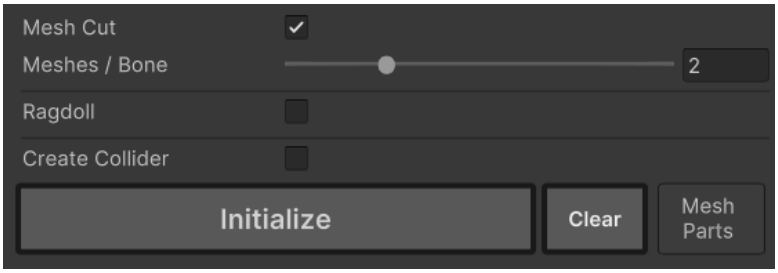
Puppet Master

If you want to use Puppet Master instead of the integrated ragdoll solution, please follow this guide:

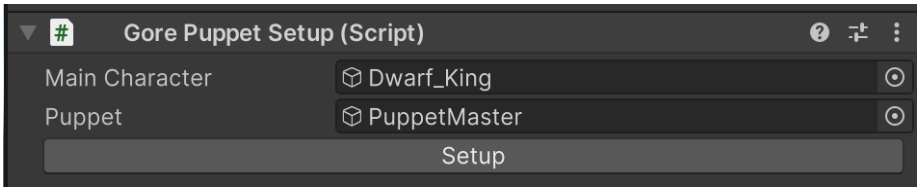
- First, create the puppet for the Puppet Master component according to the root-motion documentation and ensure that everything works as it's supposed to.

- Assign the Gore Simulator component to your main character (not the puppet) and also assign the bones from the main character.

- Before initialization, ensure that both 'Ragdoll' and 'Create Collider' options are unchecked, and then initialize.



- Attach the ‘GorePuppetSetup’ script to any GameObject.
- Assign the Gore Simulator character and the Puppet Master puppet and click ‘Setup’.

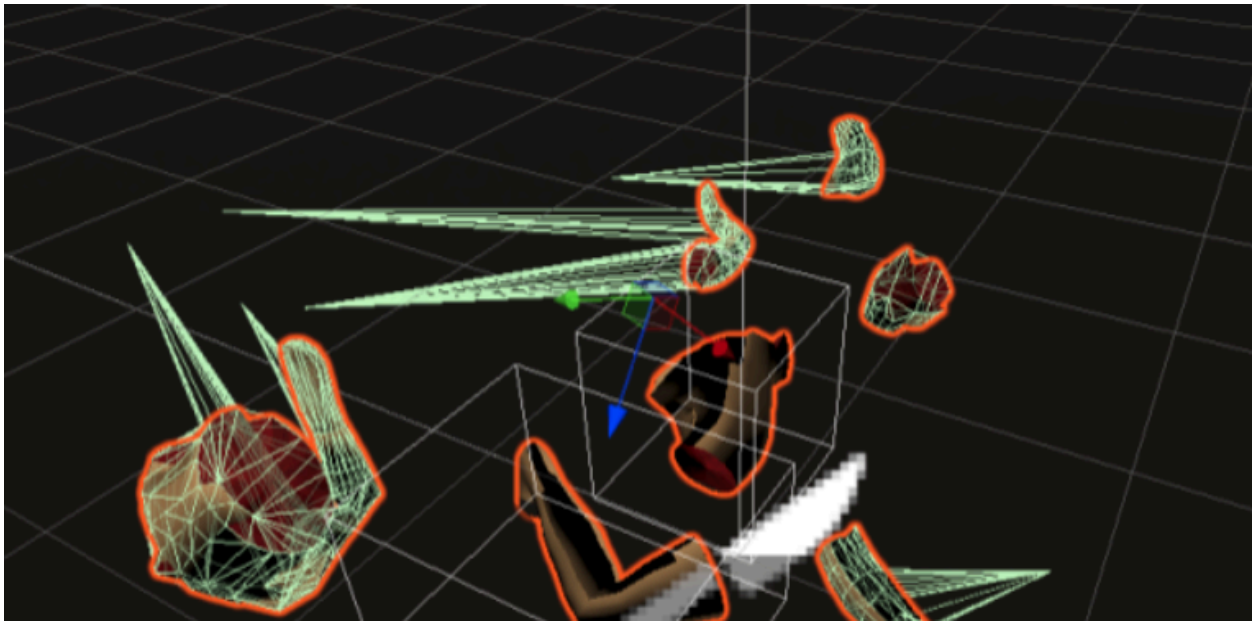


That should be it. Now, executing Gore Simulator, for instance with the included demo scripts, should be straightforward, given that the 'GorePuppet' script attached to the puppet bones implements the 'IGoreObject' interface.

TROUBLESHOOTING

Distorted Mesh

You might come across issues with detached parts resembling those shown in this image:



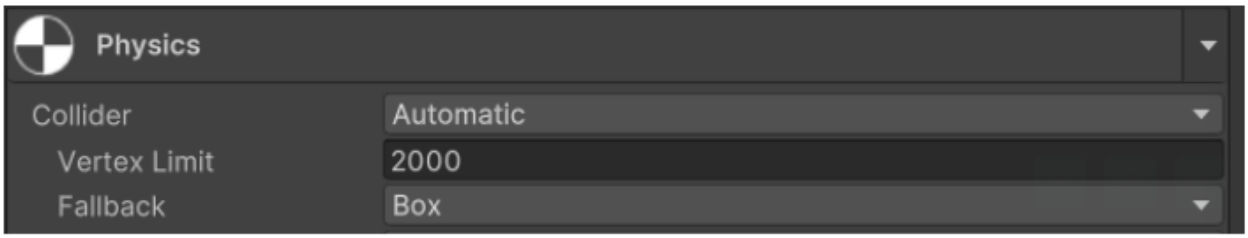
This is usually the result of runtime mesh creation for more complicated meshes. This issue can be addressed by enabling 'Optimize' in the Physics module.

Convex Mesh Warning

When adding Mesh Colliders to detached parts, Unity may display a warning message:

'Couldn't create a Convex Mesh from source mesh within max polygon limit. Partial hull used.'

To resolve this issue, you can opt for 'Automatic' colliders and gradually reduce the 'Vertex Limit' until the warning no longer appears for this character.



Scene Cleanup

By default, Gore Simulator utilizes pooling for detached parts and effects. By default, pooled objects are hidden in the editor. If you wish to track pooled objects, you can unhide them through the Global Settings.
If performance is not a concern, pooling can also be deactivated in the Global Settings.

When the internal pool is in use, scenes may contain active pooled objects that need proper release before unloading.

Typically, this is done automatically by the tool. If you still encounter a log error message from Gore Simulator mentioning pooling, or if you encounter the following error:

```
Some objects were not cleaned up when closing the scene. (Did you spawn new
GameObjects from OnDestroy?)
The following scene GameObjects were found:
...
```

You can resolve it by either call:

```
goreSimulator.SceneCleanup();
```

on each active Gore Simulator in that scene, or simply:

```
GoreSimulatorAPI.SceneCleanup();
```

to clean up all active Gore Simulators before closing a scene.

API

IGoreObject.cs

Cut and explosion operations should preferably be executed over the IGoreObject which is attached to each of the assigned bones, since this interface covers both intact and cut bones.

Some examples can be found in the provided demo scenes.

```
public void ExecuteCut(Vector3 position);

public void ExecuteCut(Vector3 position, Vector3 force);

public void ExecuteCut(string boneName, Vector3 position);

public void ExecuteCut(string boneName, Vector3 position, Vector3
force);

public void ExecuteCut(Vector3 position, out GameObject
detachedObject);
public void ExecuteCut(Vector3 position, Vector3 force, out
GameObject detachedObject);

public void ExecuteExplosion();

public void ExecuteExplosion(float radialForce);

public void ExecuteExplosion(Vector3 position, float force);

public void ExecuteExplosion(out List<GameObject> explosionParts);

public void ExecuteExplosion(float radialForce, out List<GameObject>
explosionParts);

public void ExecuteExplosion(Vector3 position, float force, out
List<GameObject> explosionParts);

public void SpawnCutParticles(Vector3 position, Vector3 direction);
```

GoreSimulator.cs

Public Actions.

```
public event Action OnExecute;

public event Action<string> OnExecuteCut;

public event Action OnExecuteExplosion;

public event Action OnExecuteRagdoll;

public event Action OnCharacterReset;

public event Action OnDeath;

public event Action OnDestroyAction;
```

```
/// <summary>
///     Executes a cut at the bone nearest to the given position.
///     Please note that this method incurs some overhead due to the
bone-finding process.
/// </summary>
/// <param name="position">The position at which the cut should be
executed.</param>
public void ExecuteCut(Vector3 position)


/// <summary>
///     Executes a cut on the mesh for the specified bone.
/// </summary>
/// <param name="boneName">Name of the bone that should be
cutted.</param>
/// <param name="position">Position of the cut. Finds the nearest
possible mesh part of that bone.</param>
public void ExecuteCut(string boneName, Vector3 position)


/// <summary>
///     Executes a cut at the bone nearest to the given position.
///     Please note that this method incurs some overhead due to the
bone-finding process.
/// </summary>
/// <param name="position">The position at which the cut should be
executed.</param>
/// <param name="force">Force to be applied to the cut mesh.</param>
public void ExecuteCut(Vector3 position, Vector3 force)


/// <summary>
///     Executes a cut on the mesh for the specified bone.
/// </summary>
/// <param name="boneName">Name of the bone that should be
cutted.</param>
/// <param name="position">Position of the cut. Finds the nearest
possible mesh part of that bone.</param>
/// <param name="force">Force to be applied to the cut mesh.</param>
public void ExecuteCut(string boneName, Vector3 position, Vector3
force)
```

```
/// <summary>
///     Executes a cut on the mesh for the specified bone.
/// </summary>
/// <param name="boneName">Name of the bone that should be
cut.</param>
/// <param name="position">Position of the cut. Finds the nearest
possible mesh part of that bone.</param>
/// <param name="detachedObject">The detached object in the scene
which has the different mesh parts as its children.
/// This object should not be destroyed but returned to the pool using
the ResetCharacter() or DespawnDetachedObjects() method.
/// Alternatively, the 'Auto Despawn' module could be used, or the
character itself could simply be destroyed.</param>
public void ExecuteCut(string boneName, Vector3 position, out
GameObject detachedObject)


public void ExecuteCut(string boneName, Vector3 position, Vector3
force, out GameObject detachedObject)


public void ExecuteCut(Vector3 position, out GameObject
detachedObject)
```

```
/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
public void ExecuteExplosion()


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="radialForce"> Radial force to be added to the
explosion parts, directed from the character center to each cutted
part.
/// Note: Requires the physics submodule with rigidbody checked.
</param>
public void ExecuteExplosion(float radialForce)


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="position">Explosion center from where a spherical
force is applied on the mesh parts.</param>
/// <param name="force"> Force to be added to the explosion parts,
directed from the position to each cutted part.
/// Note: Requires the physics submodule with rigidbody checked.
</param>
public void ExecuteExplosion(Vector3 position, float force)


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="explosionParts">A list of the explosion parts in the
scene.
/// These parts should not be destroyed but returned to the pool using
the ResetCharacter() or DespawnDetachedObjects() method.
/// Alternatively, the 'Auto Despawn' module could be used, or the
character itself could simply be destroyed.</param>
public void ExecuteExplosion(out List<GameObject> explosionParts)
```

```
/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="radialForce"> Radial force to be added to the
explosion parts, directed from the character center to each cutted
part.
/// Note: Requires the physics submodule with rigidbody checked.
</param>
/// <param name="explosionParts">A list of the explosion parts in the
scene.
/// These parts should not be destroyed but returned to the pool using
the ResetCharacter() or DespawnDetachedObjects() method.
/// Alternatively, the 'Auto Despawn' module could be used, or the
character itself could simply be destroyed.</param>
public void ExecuteExplosion(float radialForce, out List<GameObject>
explosionParts)


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="position">Explosion center from where a spherical
force is applied on the mesh parts.</param>
/// <param name="force"> Force to be added to the explosion parts,
directed from the position to each cutted part.
/// Note: Requires the physics submodule with rigidbody checked.
</param>
/// <param name="explosionParts">A list of the explosion parts in the
scene.
/// These parts should not be destroyed but returned to the pool using
the ResetCharacter() or DespawnDetachedObjects() method.
/// Alternatively, the 'Auto Despawn' module could be used, or the
character itself could simply be destroyed.</param>
public void ExecuteExplosion(Vector3 position, float force, out
List<GameObject> explosionParts)
```

```
/// <summary>
///     Activates ragdoll mode for the character. Ensure that the
Unity Animator component is deactivated and any
///     custom character controllers do not interfere with the
physics-based ragdoll behavior.
/// </summary>
public void ExecuteRagdoll()


/// <summary>
///     Activates ragdoll mode for the character. Ensure that the
Unity Animator component is deactivated and any
///     custom character controllers do not interfere with the
physics-based ragdoll behavior.
/// </summary>
/// <param name="force">Adds a force in world space to the center
bone.</param>
public void ExecuteRagdoll(Vector3 force)


/// <summary>
///     Activates ragdoll mode for the character. Ensure that the
Unity Animator component is deactivated and any
///     custom character controllers do not interfere with the
physics-based ragdoll behavior.
/// </summary>
/// <param name="force">Adds a force in world space to the specified
bone.</param>
/// <param name="boneName">Name of the bone to add the force
to.</param>
public void ExecuteRagdoll(Vector3 force, string boneName)
```

```
/// <summary>
///     Activates ragdoll mode for the character and executes a cut at
the specified position.
/// </summary>
/// <param name="boneName">Name of the bone that should be
cut.</param>
/// <param name="position">Position of the cut. Finds the nearest
possible mesh part of that bone.</param>
/// <param name="force">Force to be applied to the cut mesh and
ragdoll bone.</param>
public void ExecuteRagdollCut(string boneName, Vector3 position,
Vector3 force)


/// <summary>
///     Resets the character back to its initial state.
/// </summary>
public void ResetCharacter()


/// <summary>
///     Spawns particles from the Cut SubModuleParticleEffects module
without executing a cut.
/// </summary>
public void SpawnCutParticles(Vector3 position, Vector3 direction)
```

```
/// <summary>
///     Destroys this Gore Simulator component and despawns associated
pooled objects.
///     You can also call <see cref="GoreSimulatorAPI.SceneCleanup"/>
to clean up all active Gore Simulator components at once.
/// </summary>
public void SceneCleanup()


/// <summary>
///     Despawns all active objects that have been created by this
component in cut or explosion operations, recycling them into the
object pool.
/// </summary>
public void DespawnDetachedObjects()


/// <summary>
///     Records the current hierarchy.
///     Required if GameObjects added to the bone hierarchy after game
start should be registered.
/// </summary>
public void RecordHierarchy()


/// <summary>
///     Restores the last recorded hierarchy. By default, the
hierarchy is being recorded once in Awake().
///     This method is also being called automatically with <see
cref="ResetCharacter" />.
/// </summary>
public void RestoreHierarchy()


/// <summary>
///     Get all active GameObjects in the scene that have been created
by this component.
/// </summary>
public List<GameObject> GetCreatedObjects()


/// <summary>
///     Retrieves all detached children of active Gore Simulators.
/// </summary>
public static List<GameObject> GetAllDetachedChildren()
```

```
/// <summary>
///     Get all active Particle Systems in the scene that have been
created by this component.
/// </summary>
public List<GameObject> GetActiveParticles()


/// <summary>
///     Get all active Particle Systems in the scene that have been
created by the Cut Module.
/// </summary>
public List<GameObject> GetActiveCutParticles()


/// <summary>
///     Get all active Particle Systems in the scene that have been
created by the Explosion Module.
/// </summary>
public List<GameObject> GetActiveExplosionParticles()


/// <summary>
///     Get a main gore module from this component.
/// </summary>
public T GetGoreModule<T>() where T : GoreModuleBase


/// <summary>
///     Get a cut sub-module from this component.
/// </summary>
public T GetSubModuleCut<T>() where T : SubModuleBase


/// <summary>
///     Get an explosion sub-module from this component.
/// </summary>
public T GetSubModuleExplosion<T>() where T : SubModuleBase


/// <summary>
///     Get a ragdoll sub-module from this component.
/// </summary>
public T GetSubModuleRagdoll<T>() where T : SubModuleBase
```

```
/// <summary>
///     Activates/deactivates a sub-module in the <see
cref="GoreModuleCut"/> module.
/// </summary>
/// <param name="index">Index of the submodule.</param>
/// <param name="active">Set active or inactive.</param>
public void SetSubModuleCutActive(int index, bool active)


/// <summary>
///     Activates/deactivates a sub-module in the <see
cref="GoreModuleExplosion"/> module.
/// </summary>
/// <param name="index">Index of the submodule.</param>
/// <param name="active">Set active or inactive.</param>
public void SetSubModuleExplosionActive(int index, bool active)


/// <summary>
///     Activates/deactivates a sub-module in the <see
cref="GoreModuleRagdoll"/> module.
/// </summary>
/// <param name="index">Index of the submodule.</param>
/// <param name="active">Set active or inactive.</param>
public void SetSubModuleRagdollActive(int index, bool active)


/// <summary>
///     Overwrites the material colors for this component.
/// </summary>
public void SetComponentColor()


/// <summary>
///     Overwrites the material colors for this component.
/// </summary>
/// <param name="color">Color value.</param>
public void SetComponentColor(Color color)
```

GoreSimulatorAPI.cs

```
/// <summary>
///     Get a list of all active <see cref="GoreSimulator"/>s.
/// </summary>
public static List<GoreSimulator> GetActiveGoreSimulators()


/// <summary>
///     Returns the total number of active <see
cref="GoreSimulator"/>s in the hierarchy.
/// </summary>
public static int ActiveGoreSimulators()


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
public static void ExecuteExplosionAll()


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="radialForce"> Radial force to be added to the
explosion parts, directed from the character center to each cutted
part.
/// Note: Requires the physics submodule with rigidbody checked.
</param>
public static void ExecuteExplosionAll(float radialForce)


/// <summary>
///     Executes an explosion on the mesh.
/// </summary>
/// <param name="position">Explosion center from where a spherical
force is applied on the mesh parts.</param>
/// <param name="force"> Force to be added to the explosion parts,
directed from the position to each cutted part.
/// Note: Requires the physics submodule with rigidbody checked.
</param>
public static void ExecuteExplosionAll(Vector3 position, float force)
```

```
/// <summary>
///     Activates ragdoll mode for the character. Ensure that the
Unity Animator component is deactivated and any
///     custom character controllers do not interfere with the
physics-based ragdoll behavior.
/// </summary>
public static void ExecuteRagdollAll()


/// <summary>
///     Cleans up all active Gore Simulator components and associated
pooled objects in every scene.
/// </summary>
public static void SceneCleanup()


/// <summary>
///     Despawns all active objects that have been created by Gore
Simulators.
/// </summary>
public static void DespawnAllDetachedObjects()


/// <summary>
///     Resets all characters back to their initial state.
/// </summary>
public static void ResetCharacters()


/// <summary>
///     Records the current hierarchy of all Gore Simulators.
///     Required if GameObjects added to the bone hierarchy after game
start should be registered.
/// </summary>
public static void RecordHierarchies()


/// <summary>
///     Restores the last recorded hierarchy of all Gore Simulators.
By default, the hierarchy is being recorded once in Awake().
///     This method is also being called automatically with <see
cref="ResetCharacter" />.
/// </summary>
public static void RestoreHierarchies()
```

```
/// <summary>
///     Get all active GameObjects in the scene that have been created
by Gore Simulator components.
/// </summary>
public static List<GameObject> GetAllCreatedObjects()


/// <summary>
///     Retrieves all detached children of active Gore Simulators.
/// </summary>
public static List<GameObject> GetAllDetachedChildren()


/// <summary>
///     Get all active Particle Systems in the scene that have been
created by Gore Simulators.
/// </summary>
public static List<GameObject> GetAllActiveParticles()


/// <summary>
///     Get all active Particle Systems in the scene that have been
created by the Cut Modules from Gore Simulators.
/// </summary>
public static List<GameObject> GetAllActiveCutParticles()


/// <summary>
///     Get all active Particle Systems in the scene that have been
created by the Explosion Module from Gore Simulators.
/// </summary>
public static List<GameObject> GetActiveExplosionParticles()


/// <summary>
///     Overwrites the material colors for all Gore Simulators.
/// </summary>
public static void SetComponentColors()


/// <summary>
///     Overwrites the material colors for all Gore Simulators.
/// </summary>
/// <param name="color">Color value.</param>
public static void SetComponentColors(Color color)
```