Sakari Malkki

Painting on 3D Objects During a Game’s Runtime

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology,

Bachelor’s Thesis

2 March 2023

Abstract

Author:	Sakari Malkki
Title:	Painting on 3D objects and characters during a game's runtime
Number of Pages:	53 pages
Date:	2 March 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Game Applications
Supervisors:	Senior Lecturer Antti Laiho

Unity game engine comes with some tools which can be used for painting on objects and characters during a game's runtime. These features work well for some use cases, usually when involving static 3D objects. More specific situations, especially when skinned and animated character are involved, require more intricate solutions.

Such features are central part of making games feel responsive and dynamic, which directly adds to the game feel, often referred to as "juiciness" in the game industry. The context of this study, a game called Para Bellum - Frontlines has a few different specified use cases for altering objects' surface for both static models and animated characters such as bullet holes on objects, impact wounds on characters and blood splatters and pools in the environment. Such features are often referred to as response mechanics.

The goal of this study was to find the most efficient solutions for the specified use cases in the game's context, while also trying to implement generic enough solutions for future use in other projects. Such features could help with improving the game feel and responsiveness in the future with easy reimplementation for new projects.

A couple of different solutions were identified and implemented for the use cases, including the use of Unity's Universal Render Pipeline compatible Decal Renderers, shaders, runtime texture modification and a third-party Skinned Mesh Decals plugin.

The implemented solutions turned out to work well for their intended purposes and are generic enough for repurposing for future projects. Bullet holes, blood splatters and blood pools were implemented using Unity's Decal Renderers. Impact wounds utilize Render Textures and a third-party plugin called Skinned Mesh Decals and generic blood overlays were implemented using a combination of custom shaders and runtime texture modification. While there is still room for improvement when it comes to fixing edge cases and improving runtime performance, the results are good and appear to be useful for future purposes as well.

Keywords: decals, shaders, materials, technical art, textures, projectors,

Tiivistelmä

Tekijä:	Sakari Malkki
Otsikko:	Reaaliaikainen maalaaminen pelin 3D-objekteihin
Sivumäärä:	53 sivua
Aika:	2.3.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Antti Laiho

Unity-pelimoottori tarjoaa tietyt työkalut, joilla peliobjekteja tai -hahmoja voidaan maalata pelissä. Nämä työkalut toimivat hyvin joissakin yksinkertaisissa tarkoituksissa silloin, kun kohteena ovat staattiset 3D-objektit. Monimutkaisemmat tarpeet, erityisesti silloin, kun kohteena ovat skinnatut ja animoidut 3D-hahmot, vaativat työläämpiä ratkaisuja.

Peliobjekteihin ja -hahmoihin maalaaminen on hyvin keskeinen ominaisuus, kun peleistä halutaan tehdä responsiivisen ja dynaamisen tuntuisia, joilla suoraan vaikutetaan pelikokemukseen. Tämän tutkimuksen kontekstina toimiva peli Para Bellum - Frontlines tarvitsee tietyt toiminnot, joissa tarvitaan peliobjekteihin tai hahmoihin maalaamista. Esimerkkejä näistä ominaisuuksista ovat luodinreiät staattisissa peliobjekteissa ja veriroiskeet ja lammikot pelin ympäristöissä.

Insinöörityön tarkoituksena oli selvittää tehokkaimmat ratkaisut pelin tarpeiden mukaisesti ja implementoida ne käyttäen erilaisia teknologioita ja ratkaisuja. Lisäksi tarkoituksena oli tehdä implementaatioista niin yleiskäyttöisiä, että niitä voidaan käyttää myös tulevissa projekteissa suhteellisen pienellä vaivalla.

Eri käyttötarkoituksia varten tehtiin muutama eri ratkaisu. Niihin käytettiin Unityn Universal Render Pipelinen kanssa yhteensopivaa Decal Renderer -ominaisuutta, shadereita, tekstuurien reaaliaikaista muokkausta ja kolmannen osapuolen tarjoamaa Skinned Mesh Decals -työkalua.

Valmiit ratkaisut osoittautuivat toimiviksi niiden käyttötarkoituksia ajatellen, ja niiden implementointi onnistui siinä määrin yleiskäyttöisesti, että niitä voidaan suhteellisen vaivattomasti käyttää tulevissa projekteissa ja eri käyttötarkoituksissa. Luodinreiät, veriroiskeet ja -lammikot toteutettiin käyttämällä Unityn Decal Renderer -ominaisuutta, ampumahaavoihin käytettiin Unity Render Textureita sekä kolmannen osapuolen työkalua nimeltä Skinned Mesh Decals. Veriteksturointi tehtiin käyttämällä shadereitä ja tekstuurien reaaliaikaista modifiointia. Vaikka ratkaisuihin jäi vielä parantamisen varaa esimerkiksi performanssin ja erityisolosuhteiden aiheuttamien ongelmien kanssa, tulokset olivat lupaavia ja hyöty tulevien projektien kannalta ilmeinen.

Avainsanat: decalit, shaderit, materiaalit, technical art, tekstuurit, projektorit

Contents

List of Abbreviations

1 Introduction – Game Feel and Response Mechanics 1

2 Context and Use Cases 2

2.1 Para Bellum - Frontlines 2
2.2 Bullet Holes on Static Objects 4
2.3 Blood Splatters and Blood Pools 5
2.4 Impact Wounds on Characters 7
2.5 Generic Bloody Overlay on Characters 9

3 Tools and Technologies Used for Painting on GameObjects 10

3.1 Unity Engine 10
3.2 Universal Render Pipeline (URP) 11
3.3 Shaders 11
3.4 Decal Renderer 11
3.5 Shader Graph 12
3.6 Runtime Texture Modifying 13
3.7 Render Textures 13
3.8 Skinned Mesh Decals 13

4 Solutions for Painting Use Cases 14

4.1 Bullet Holes on Static Objects 14
4.2 Blood Splatters and Blood Pools 21
4.3 Impact Wounds on Characters 30
4.4 Generic Blood Overlay 39

5 Conclusion 44

5.1 Results and Analysis 44
5.2 Future Considerations 45

References 1

List of Abbreviations

URP:	Universal Render Pipeline. Unity’s scriptable render pipeline for multi-platform use. Also known as Lightweight Render Pipeline (LWRP).
HLSL:	High Level Shader Language. Programming language for shader programming.
PBR:	Physically Based Renderer. Real time 3D rendering method that simulates the way light works in real life with properties such as metallic, smoothness, roughness, and normal maps.
GPU:	Graphics processing unit. Computer’s hardware computer primarily responsible for rendering graphics on the computer’s screen.
CPU:	Central processing unit. Computer’s main processor.

1 Introduction – Game Feel and Response Mechanics

Painting on 3D game objects during a game’s runtime, meaning altering the appearance of a game object’s surface material and textures is a very common feature that can be used for a multitude of use cases in all kinds of video games. Such examples could include for example customizing a player’s car in a car racing game with decals and custom colours, or to provide responsive feedback for player’s actions in a visual manner in action games.

Response mechanics in games are mechanics and systems that provide dynamic and emergent feedback to the player for their actions. Such features improve the gameplay feel – often referred to as “game juice” in the industry (1) – and make the game feel more responsive and immersive.

Since games are an audio-visual form of media, response mechanics take the form of different audible and visual events and effects that happen in response to the player’s actions. Simple examples of such mechanics could include a sound effect played when a player collects a coin or a particle effect that shows that a player did damage to an enemy with their attack. Game responsiveness and feel can be improved in more technical and subtle ways too such as adding a slight shake to the game’s camera on explosions and other impactful events, or by allowing the player to destroy their surroundings such as buildings that are in the game world.

Painting on 3D game objects is a very commonly used technique for game response mechanics. Examples of such mechanics take place in virtually all shooter and action games nowadays. (2) Common examples of that are for example bullet holes in objects that the player shoots or burn marks on floors after explosions. While the mentioned examples include mostly static objects and are usually easier to implement, there are also common examples that involve painting on animated 3D characters that are often more technically involved. Examples of such se cases include leaving wounds on characters shot by the player or otherwise damages or having a bloody or burnt overlay on characters that have been killed by fire or explosions. Such systems are often referred to as gore mechanics. Less violent examples of such systems could include player character’s feet and legs getting wet or muddy after waddling through a puddle of dirt or water.

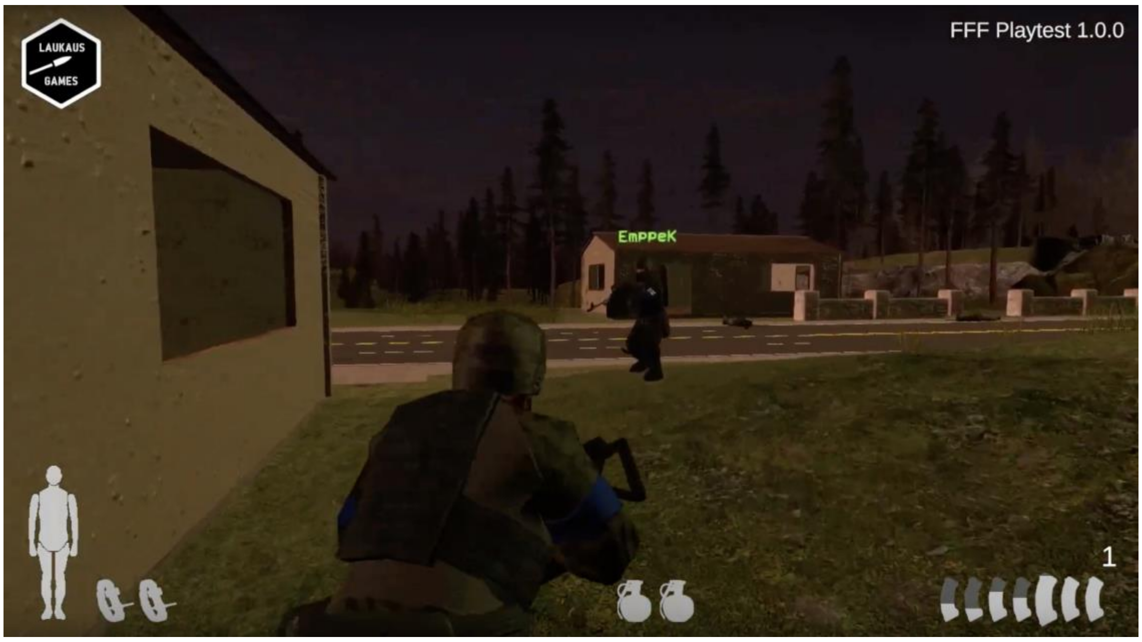
Unity game engine provides some out-of-the-box solutions for these purposes, but those options are somewhat limited especially when it comes to modifying the textures and materials of animated and skinned 3D meshes. The goal of this study is to come up with some common use cases for painting on 3D objects and characters in a Unity game in a context of one specific game, and to find efficient and generic enough technical solutions that can be used not only in the context of that game, but also in future projects.

2 Context and Use Cases

2.1 Para Bellum - Frontlines

The context of this thesis study is a game project called Para Bellum – Frontlines made by Laukaus Games. The game is a 3D third person shooter action game that can be played either alone in single player or cooperatively with other players online. The game is military themed and features instant fast paced action from old school action shooters blended with modern features from tactical shooters such as realistic damage and weapon handling. A public demo of the game is available to download for free on Windows, Mac OS, and

Linux platforms on Itch.io. (3) Below in Picture 1 is a screenshot from the game’s early playtest.



Picture 1: Screenshot from Para Bellum – Frontline’s first official playtest.

The game has strong emphasis on game feel and responsiveness. The goal is to have the game environment and characters react in a satisfying manner to all player and non-player-character actions. While the game is not mechanically very deep, a large part of its enjoyability in its design comes down to how responsive and juicy it feels to play. Among other response mechanics, painting on objects, both static and animated in runtime is crucial for providing responsive feedback for player’s actions. Player’s actions in this regard mostly revolve around using different weapons and explosives. Response both from the game environment and enemy characters needs to present this accordingly. The plan is that whatever it is that the player shoots, would react in a manner that the player expects. Some of these reactions require solutions such as particle effects and destruction mechanics, but for most of the cases, just changing the object's or character’s texture or material is going to go a long way.

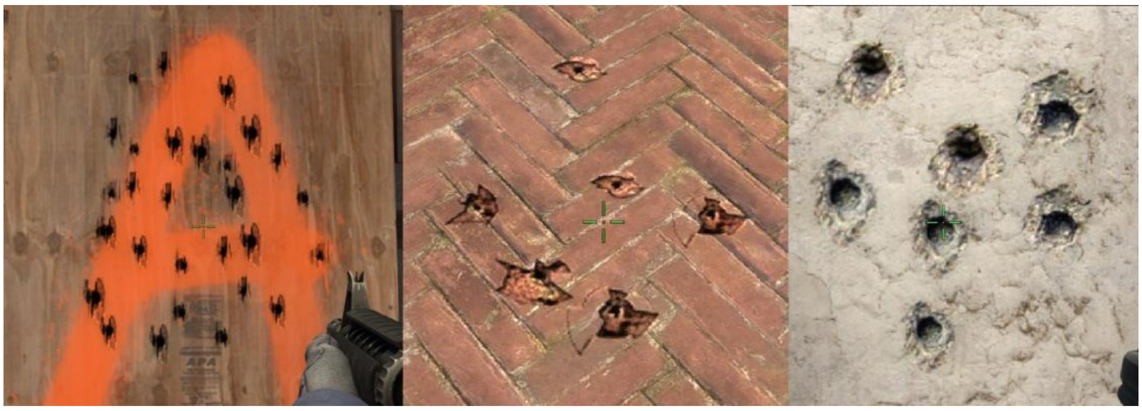
The goal of this study is to recognize the use cases for which runtime painting on 3D characters and objects can be used to improve the feel of the game and to come up with most suitable and efficient technical solutions for those features. Since the game is being worked on in the Unity engine, Unity engine's capabilities for delivering such responses was chosen as the subject of this thesis.

2.2 Bullet Holes on Static Objects

Bullet holes in shooter games are a very common feature. When a player or a non-player character shoots at an object in the game world, a bullet hole appears at the point of impact. The simplest implementation of the feature is just a generic black sprite rendered on a quad object that is positioned and orientated at the point of impact based on the hit surface normal. The problem with this arises when a bullet hole is shot at the edge of a wall for example and half of the bullet hole appears to be floating in the air (4).

The feature can be however implemented with much more technical fidelity. In most modern games the decal is projected on the surface in a way where it follows its geometry and instead of being just a generic texture drawn on the hit surface, the bullet hole can affect other properties of the hit object as well such as its normal map. Additionally, there can be many different variations of bullet holes for different surface materials, the decal can be randomly rotated or scaled, or the scale can be affected by the weapon used for example.

For this project, the goal of the feature was following: When a player or a non-player character shoots at an object in the game world, a bullet hole should appear at the point of impact. The bullet hole's appearance should be dependent of the material the hit object is made of. A bullet hole in a metal beam for example should look different from a bullet hole on a wooden wall. Bullet holes should affect the hit surface's albedo and normal map properties to give a convincing impression of a hole on the surface in addition to just a colour change. Additionally, the rotation of each bullet hole should be randomized, and the size of each bullet hole should be varied slightly to avoid the visuals becoming too repetitive. The closest reference feature was found from Valve's 2012 game Counter Strike: Global Offensive, which is shown in the picture below (Picture 2).



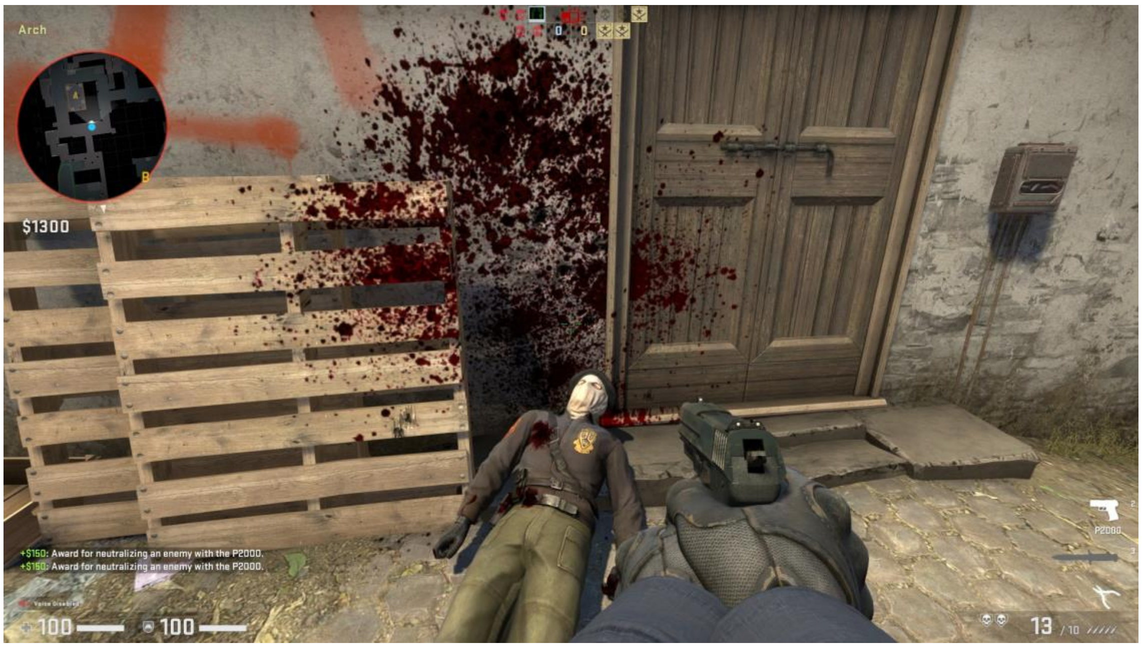
Picture 2: Examples of bullet holes on varying surface materials from game Counter Strike: Global Offensive (Valve, 2012)

As seen in the picture above (Picture 2), Counter Strike: Global Offensive’s bullet holes suit most of the demands above, they appear different on different surfaces, there’s a few different variations and bullet holes are randomly rotated and scaled.

2.3 Blood Splatters and Blood Pools

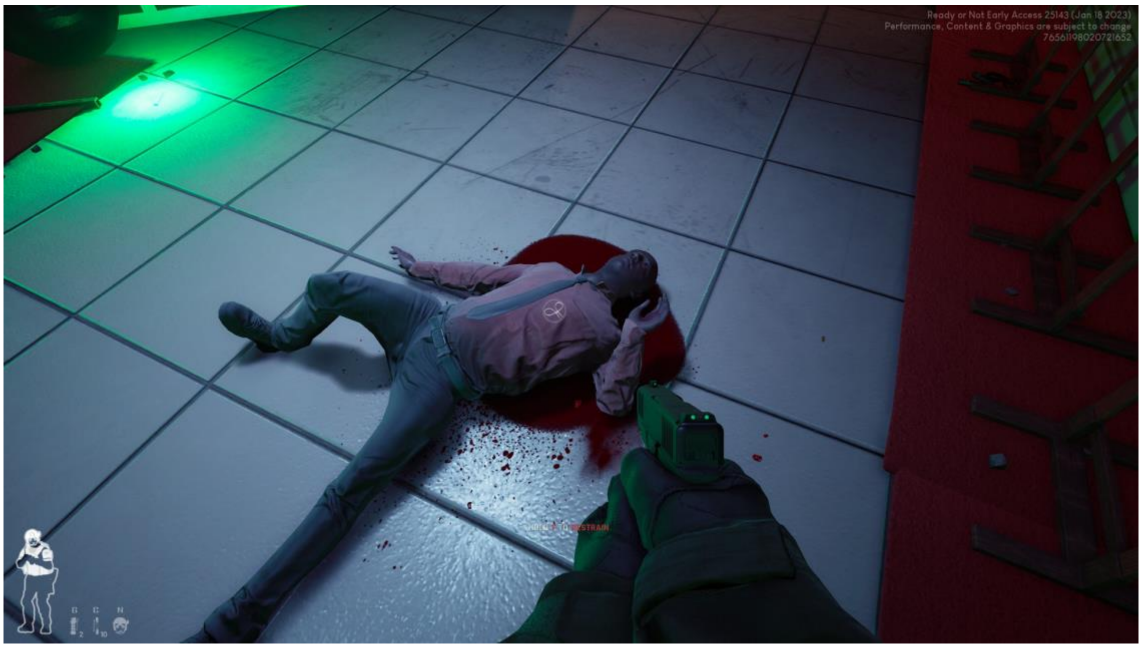
When a player or a non-player-character gets shot, it should be reflected in the surrounding environment by spawning a blood splatter decal. That decal should emit to the direction of the shot simulating a bullet going through the character and sputtering blood from the exit wound. If there’s a surface directly behind the hit character, the blood splatter should be projected on that. In case there’s no surface behind the character at a reasonable distance, the blood splatter should appear on the ground instead. The blood splatter needs to support randomly varied size and rotation. There also needs to be support for varying textures to avoid too much repetition.

Again, the reference for this feature was found from Counter Strike: Global Offensive. As shown in Picture 3 on the next page, the blood splatter behind the character is projected on the surfaces behind the hit character – the pallet, the wall and the door.



Picture 3: Example of blood splatter from game Counter Strike: Global Offensive (Valve, 2012) The same splatter is projected over multiple meshes and their geometries – the wall, the pallet, and the door - instead of just a flat surface.

For blood pooling mechanic, reference was found in game Ready or Not by VOID Interactive. In the picture below (Picture 4), there's an example of how a blood pool has formed under the character on the floor.



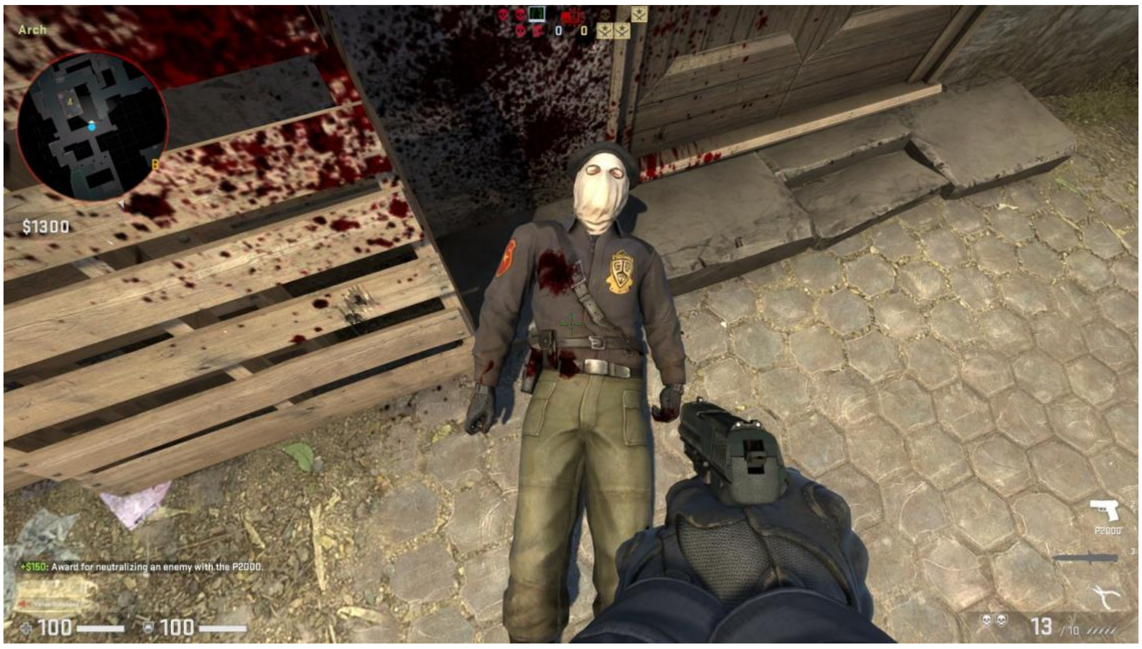
Picture 4: Blood pool from game Ready or Not (VOID Interactive, 2021). Blood pool forms slowly under a dead character spreading to different directions.

For Para Bellum – Frontlines, a blood pool should appear under a character that dies as a result of an explosion. The blood pool should form under a character and be projected on to the environment. As opposed to the blood splatters that appear immediately, blood pools should grow slowly to simulate how blood spreads on the ground.

2.4 Impact Wounds on Characters

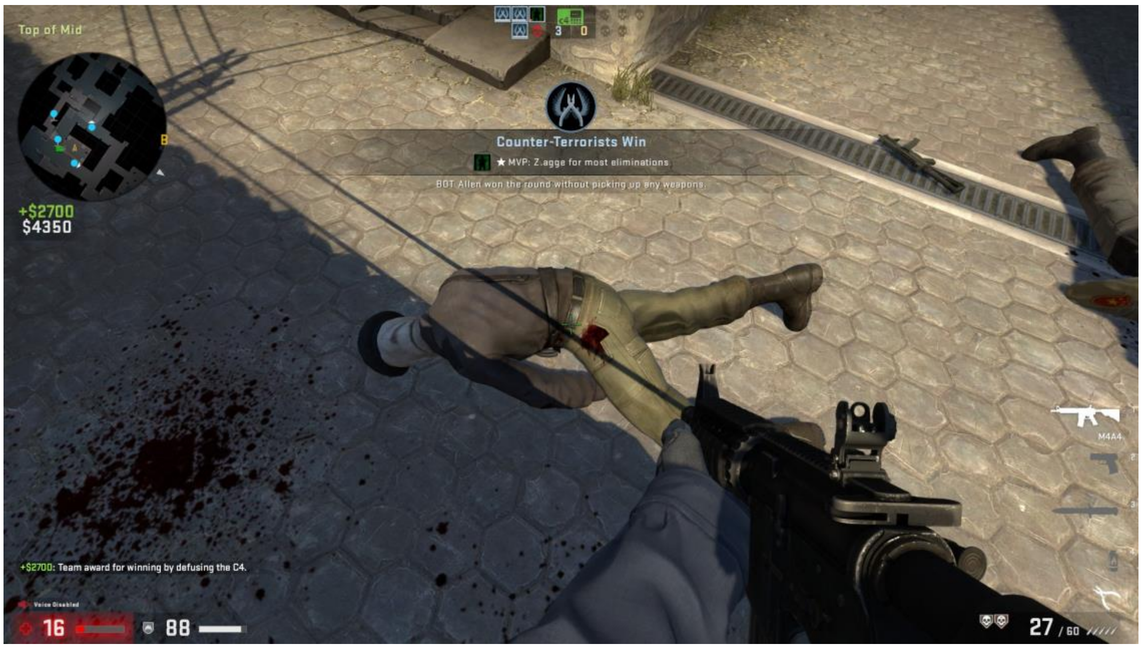
When a character is shot either by a player or a non-player-character, a wound should appear at the position of the impact. This is in other ways like the static object bullet hole feature, except for the fact that as we are now rendering a decal on an animated object's skinned 3D mesh, it is crucial to make sure that the decal works in a consistent manner on a skinned and animated mesh surface without stretching or sliding.

Counter Strike: Global Offensive has a good reference for how impact wounds on characters work as shown in Picture 5 below. The impact wound is projected at the point of impact on the hit character's chest.



Picture 5: Impact wounds on a dead character from game Counter Strike: Global Offensive (Valve, 2012)

In Source engine games such as Counter Strike: Global Offensive decals on animated characters are achieved by duplicating and extruding the surface geometry. This achieves a working result that usually looks good and works well in most cases but is prone to glitches and will result in performance issues if too many decals are drawn at the same time since the amount of geometry to draw is being increased with each new decal. (4) An example of a glitching impact wound is shown below in Picture 5. The impact wound cuts abruptly between the leg's polygons.


Top of Mid
Counter-Terrorists Win
★ MVP: Z.agge for most eliminations
BOT Allen won the round without picking up any weapons
+$2700
$4350
+$2700: Team award for winning by defusing the C4.
16
88
27 / 60
M4A4

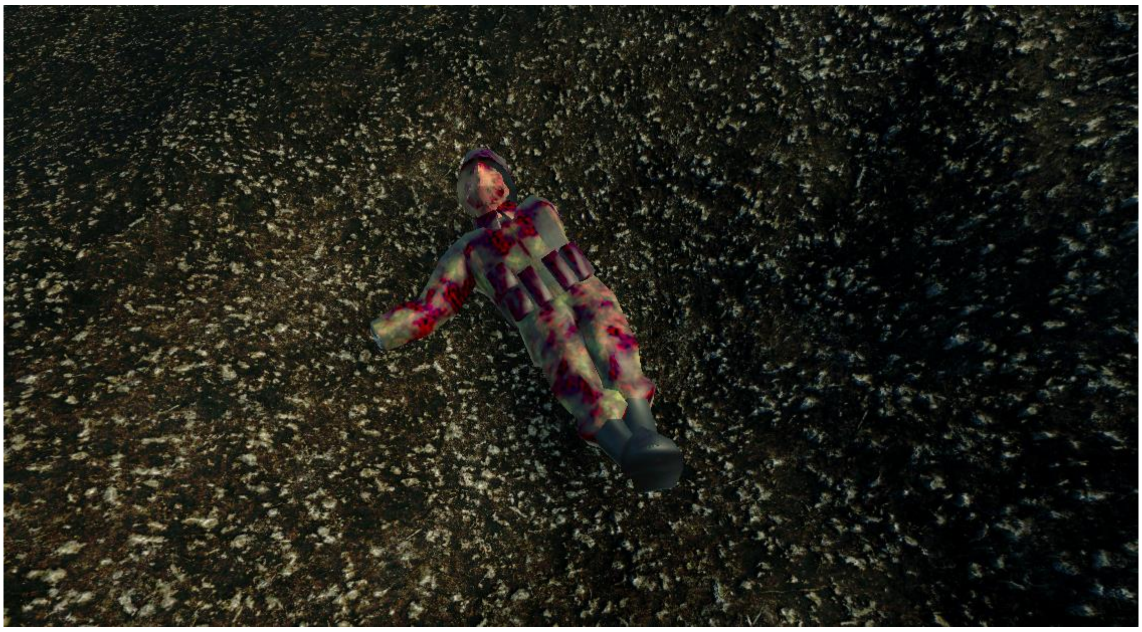
Picture 6: Example of a glitch with the Source engine's decals on characters (Counter Strike: Global Offensive, Valve, 2012). In this instance, the impact wound decal is abruptly cut between two faces of the mesh.

For Para Bellum – Frontlines an impact wound should appear on a character at the point where the shot hits. It should be projected on the character's mesh and follow the skinned mesh accurately. Impact wound sizes should be randomly varied in size and rotation. One character can have multiple impact wounds simultaneously. Impact wounds should remain on the character’s mesh indefinitely.

2.5 Generic Bloody Overlay on Characters

When a character takes damage from an explosion, there should be a bloody overlay on the affected parts of the body. This blood overlay should be randomly varied to avoid repetition and like with the impact wounds, it needs to work consistently on a skinned and animated character without stretching, sliding or other issues. In addition, it is important to have functionality to have multiple separate body parts with damage overlay visible simultaneously. For example, the character may be wounded both to their arm and leg at the same time.

It was decided to use Laukaus Games’ previous games Para Bellum – Frontlines’ solution for generic bloody overlays as a base for this feature. In that game a generic overlay of blood covers the whole character after an explosion caused death. The overlay is randomly offset, which results in varying looks between characters, avoiding repetition. An example of the game’s implementation for generic bloody overlay is shown on the next page in Picture 7.



Picture 7: Implementation of generic bloody overlay from Laukaus Games' previous game project, Para Bellum – Hold the Line. In this implementation the pattern of blood is randomized within the shader by offsetting the UV map for each instanced material. However, the overall damage is not masked, and it covers the whole character.

For Para Bellum – Frontlines the existing feature for generic blood overlay would be extended and improved to support masking on individual body parts. Additionally, there should be individual damage mask channels for partial and full damage that can be blended. This is achievable with one texture map using different colour channels for the separate masks.

3 Tools and Technologies Used for Painting on GameObjects

3.1 Unity Engine

Unity is a game engine developed by Unity Technologies that was first released in 2005 and has since been in active development. (6) Unity is free to use for beginning indie developers and students making it an easy starting point for 3D game development. Despite its low barrier of entry, Unity is also widely used in the professional video game industry. The version of Unity used for this project is the 2021.3.11f1 LTS (Long Term Support) version that was released in 2022. (7) Unity comes with certain features and tools that can be helpful when implementing solutions for painting on 3D objects in games such as Render Textures, shaders and Decal Renderers which are described below.

3.2 Universal Render Pipeline (URP)

One of the key features that the Unity 2021 LTS version enables is the Universal Render Pipeline (URP). URP is a scriptable render pipeline that has been designed to be lightweight, which ensures usability across different platforms such as mobile, consoles and PC. Universal Render Pipeline supports physically based rendering (PBR) and as extension materials that utilize normal maps and metallic and smoothness properties in addition to albedo colours. Shaders for URP can be created and edited using the Shader Graph tool. (8)

3.3 Shaders

Shaders in video games or other interactive media are computer programs that usually run on the computer's graphics processing unit (GPU). (9) Shaders can be used for example to define how an object's surface appears, post processing or other special effects. In Unity engine, shaders are written or compiled to High Level Shader Language (HLSL). (10) In addition to writing shaders in HLSL code directly a tool such as Unity's Shader Graph can be used to create and edit shaders.

3.4 Decal Renderer

Unity 2021 offers a feature that allows the use of decals using the URP. It works in a similar way as Projectors worked in previous versions of Unity using the build in legacy render pipeline. Decal Renderer projects a specific material on a surface and allows the decal to wrap around objects making it very useful for changing the surface of a material in an easy way (11). A common use case for using the Decal Renderer is to create details in the game environment such as posters or stickers on walls, cracks or dirt on ground or floor etc. Decal Renderers can be placed manually in the editor or instantiated during runtime via code commands.

Decal projectors support custom shaders which offers a lot of flexibility in customizing decals given one knows how to make shaders either in HLSL or in the Shader Graph editor. By default, Decal Renderer materials support albedo colour and normal map properties, but the shaders can be extended to also utilized smoothness and metallic values, if need be.

Decal renderer comes with certain limitations. For one, decals do not work on transparent surfaces at the time of writing this thesis. Decals also do not support render layers in the 2021 version. This has been fixed in 2022 versions, but for earlier versions this means that there’s no out-of-the-box way to define which objects should be excluded from being projected on by decals. Additionally, projectors do not work very well on skinned and animated meshes. Decals are wrapped around the mesh, but they do not consider the weight painting of the mesh, which causes a commonly known issue known as decal sliding, where the decal drawn on top of the character slides, moves and stretches around the surface, when the character is moving or otherwise animated.

3.5 Shader Graph

Shader Graph is a robust tool for Unity editor that is used for creating and editing shaders for Unity’s scriptable render pipelines – the Universal Render Pipeline or the High Definition Render Pipeline. Shader Graph is based on nodes with inputs and outputs that can be connected to each other. This makes Shader Graph very artist friendly to use and doesn’t require any specific programming skills. If the user understands the concept of shaders and basic mathematical functions, Shader Graph can deliver impressive results with an intuitive workflow and real time preview features. In addition to existing nodes, Shader Graph supports custom nodes using which the user can use additional functionality by code in their shaders. When a game is built into a standalone executable, the shaders made in Shader Graph are compiled to HLSL for better performance. (12)

3.6 Runtime Texture Modifying

Unity allows modifying texture image assets during game's runtime. This can be very useful since modified textures can then be used in a shader to determine an object's surface for example. Modified textures can be used directly as properties for a material such as albedo colour or smoothness. In other cases, a texture can be used as a mask to lerp or blend between values.

Unity has two functions that can be used for this purpose. Texture2D.SetPixel allows the user to change the colour of one pixel at a specific location at different mip levels (13). Texture2D.SetPixels takes an array of colours and changes the colours of all pixels in the given mip level of a texture. (14)

Mip levels are used for texture mipmapping. This allows for multiple versions of a texture to exist, which decrease in resolution size. The smaller resolution mip levels of a texture are used when an object is further away from the camera. This helps the game's runtime performance and often makes visuals also more readable as there's less noise and detail in further objects. (15)

3.7 Render Textures

Render Textures are texture assets that are created and can be modified during Unity game's runtime. They work in many ways in a similar manner as regular textures and some of the most common use cases are for example rendering video or game footage in the game's user interface or as textures to modify a shader during runtime like with regular textures. (16)

3.8 Skinned Mesh Decals

Skinned Mesh Decals is an open-source tool distributed on GitHub and made by a user called naelstrof (5). It is used for creating decals on skinned 3D meshes. First a decal is projected on to the character's surface. After that the character's UV is laid out to the screen space with a shader. The result is then rendered on a Render Texture. The Render Texture is then used in a shader to draw a detail map on top of the character’s base colour texture using a separate channel for UV-coordinates.

For this purpose, any 3D asset that uses Skinned Mesh Decals needs to have two separate UV-channels of which the one used for the detail map needs to be atlased meaning that there can be no overlapping UV islands to work properly. This process can be automated by using Unity’s built in lightmap generation in asset import settings, but it can also be done manually in any 3D modelling software such as Blender or 3DS Max.

While Skinned Mesh Decals is a very efficient way of drawing decals on objects CPU and GPU wise considering there’s no additional geometry drawn, it has some limitations memory usage wise since every affected asset needs a Render Texture asset for the detail map. Both the resolution size and total allocated amount of runtime memory for drawing decals can be customized by the user.

4 Solutions for Painting Use Cases

4.1 Bullet Holes on Static Objects

Spawning bullet holes on static objects was a relatively simple task to accomplish and was easiest to implement with Unity’s out-of-the-box solutions. For this purpose, Unity’s Decal Renderer feature was used. When a bullet hit is detected on a static object by a Raycast function, the game check’s whether the hit object has a HitMaterial script component attached to it. HitMaterial component is a script file that stores different material properties in to an enum variable. The proper material property for any given object can then be assigned by adding the component to the object and picking the right settings in the Unity editor’s inspector. If no HitMaterial component is found on the hit object, a generic bullet hole is used. Otherwise, the HitMaterial component’s material type enum variable is used to determine which bullet hole material and hit particle effect prefabs are used.

Here's a code sample of the HitMaterial class:

```
using System.Collections;
using UnityEngine;

public class HitMaterial : MonoBehaviour
{
// Enum type variable for determining different hit
effects. Also used for choosing the proper bullet hole
decal material.
    public ShootingEffects.HitTypes HitEffectType;
    public bool CreateBulletHoleDecal = true;
}
```

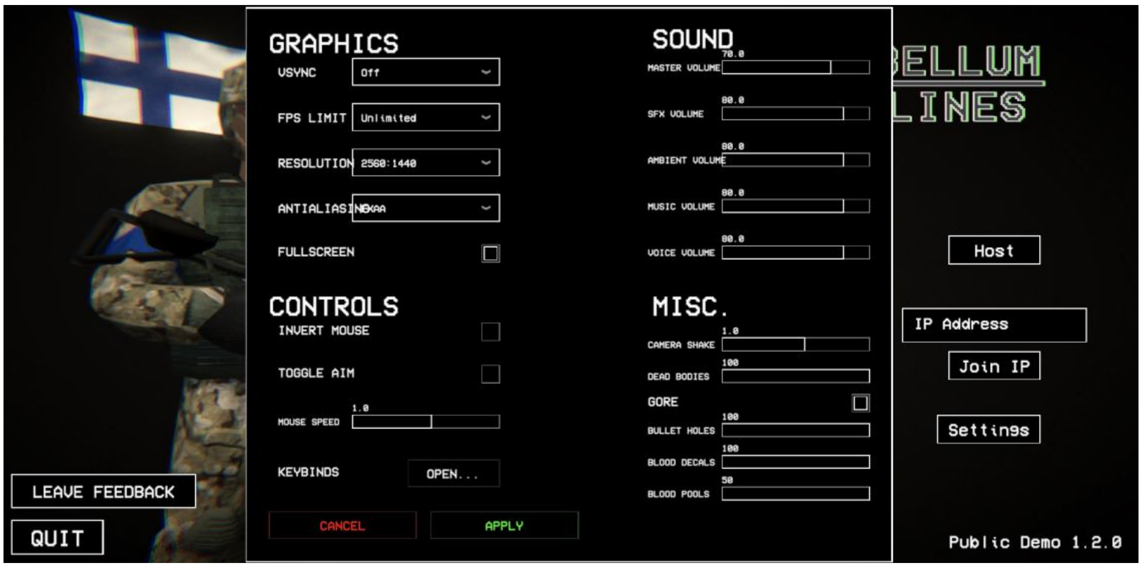
Using a Raycast function from Unity’s physics namespace allows us to store the ray’s hit location and the hit surface’s normal direction from the RaycastHit variable. (17) These are used to place and orientate both the hit particle effect and the bullet hole prefab. After a bullet hole has been instantiated, a proper material based on the HitMaterial query explained above is selected and applied to the Decal Renderer component.

the right material for the bullet hole effect and placing it is done in ShootingEffects class as follows:

```
// The function is called after a shot hit is detected in
ShootinController class. Hit position, hit surface’s normal
direction, shot direction and hit material type and Boolean
whether a bullet hole should be spawned from the
HitMaterial class are passed as arguments.
public void SpawnHitEffects(Vector3 position, Quaternion
rotation, Vector3 shootDirection, HitTypes type, bool
spawnBulletHole)
```

```
{
// Bullet hole material variable is defined. It is null if
the HitType is of type “Other” or if it has been specified
that no bullet hole should be spawned. Otherwise Generic
bullet hole material from AssetHolder singleton is used by
default.
    Material bulletHole = type == HitTypes.Other ||
!spawnBulletHole ? null :
AssetHolder.Instance.BulletHoleGeneric;

// switch-case statement for choosing the right material.
    switch (type)
    {
        case HitTypes.Metal:
            bulletHole =
AssetHolder.Instance.BulletHoleMetal;
            break;
        case HitTypes.Wood:
            bulletHole =
AssetHolder.Instance.BulletHoleWood;
            break;
        case HitTypes.Brick:
            bulletHole =
AssetHolder.Instance.BulletHoleBrick;
            break;
        case HitTypes.Rock:
            bulletHole =
AssetHolder.Instance.BulletHoleRock;
            break;
        case default:
            break;
    }
    if (bulletHole != null)
    {
// Bullet hole projector prefab is instantiated from the
pool.
        GameObject holeGO =
PoolController.Instance.PooledInstantiate(AssetHolder.Insta
nce.BulletHoleProjector.gameObject);
// Bullet hole projector transform is aligned to the shot
hit position and its surface normal rotation.
        holeGO.transform.SetPositionAndRotation(position,
rotation);
        DecalProjector projector =
holeGO.GetComponent<DecalProjector>();
// Bullet hole material is assigned to the DecalProjector
component.
        projector.material = bulletHole;
// Randomization of the bullet hole decal size.
        float scale = Random.Range(0.1f, 0.2f);
        Vector3 size = Vector3.one * scale;
        projector.size = size;
// Randomization of the bullet hole rotation.
        holeGO.transform.localEulerAngles = new
Vector3(holeGO.transform.localEulerAngles.x,
holeGO.transform.localEulerAngles.y, Random.Range(0f,
360f));
    }
}
```

To keep the game’s performance as high as possible, the player can customize a setting for how many bullet holes are visible at any given time. When the set threshold is exceeded, a pooling system will remove the oldest bullet holes and start recycling them. Doing this process known as object pooling instead of destroying objects and instantiating new ones saves CPU performance overhead. (18) A screenshot (Picture 8) from the game's settings menu below shows the settings menu for the game, from which the player can change the settings for the amounts of bullet holes. blood decals and blood pools.


GRAPHICS

VSYNC: Off
FPS LIMIT: Unlimited
RESOLUTION: 2560:1440
ANTIALIASING: FXAA
FULLSCREEN: ☐

CONTROLS

INVERT MOUSE: ☐
TOGGLE AIM: ☐
MOUSE SPEED: 1.0
KEYBINDS: OPEN...

SOUND

MASTER VOLUME: 70.0
SFX VOLUME: 80.0
AMBIENT VOLUME: 80.0
MUSIC VOLUME: 80.0
VOICE VOLUME: 80.0

MISC.

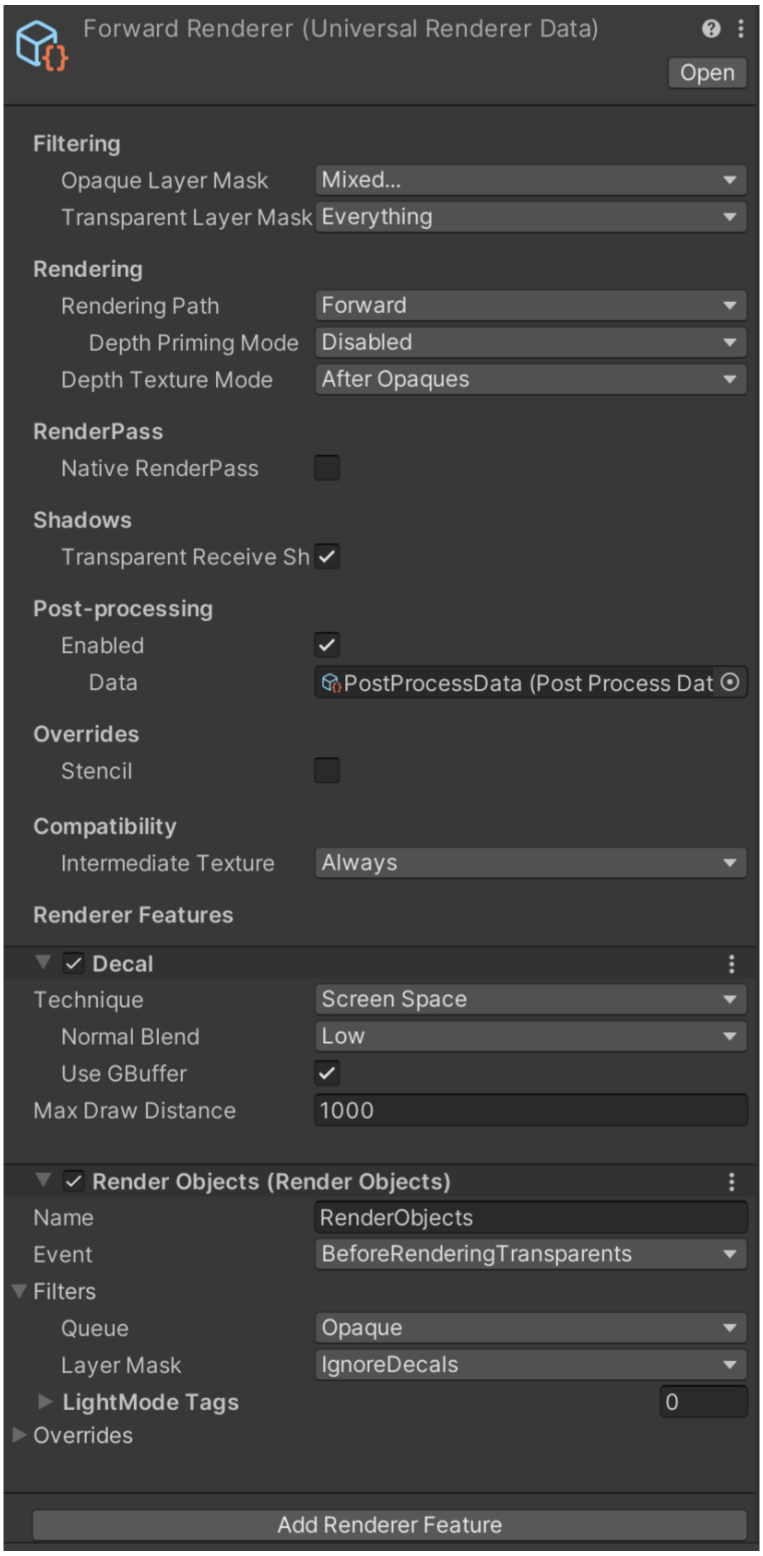
CAMERA SHAKE: 1.0
DEAD BODIES: 100
GORE: ☐
BULLET HOLES: 100
BLOOD DECALS: 100
BLOOD POOLS: 50

Picture 8: Screenshot from the game's settings menu. Under Misc. settings, the player can adjust the amounts of bullet hole, blood pool and blood splatter decals.

One shortcoming of the Decal Renderer is that in the version of Unity that is used for the project (2021.3 LTS), there's no direct solution for excluding objects being affected by the Decal Renderer. In the legacy render pipeline-based Projector, that was the previous implementation of virtually the same feature in Unity Engine, the user could set up the projector to exclude certain rendering layers. In this instance we don't want to render bullet holes or other Decal Renderers on the player characters for example.

A workaround for this issue was found by adding a rendering rule to the Universal Render Pipeline Forward Renderer settings asset for the layers that we want to exclude from the Decal Renderer. Namely in this case the IgnoreDecals layer. Layers, that we want to exclude from Decal Renderers are not drawn in the initial render pass. After the initial render pass, Decal Renderers are rendered, and the IgnoreDecals layer is rendered before rendering transparent objects. This way characters are always excluded from being drawn on by the Decal Renderers.

This also prevents us from using Decal Renderers for the character related response mechanics such as impact wounds. The setup for this is shown in the picture on the next page (Picture 9).



Forward Renderer (Universal Renderer Data)

Filtering
- Opaque Layer Mask: Mixed...
- Transparent Layer Mask: Everything

Rendering
- Rendering Path: Forward
- Depth Priming Mode: Disabled
- Depth Texture Mode: After Opaques

RenderPass
- Native RenderPass: ☐

Shadows
- Transparent Receive Sh: ✓

Post-processing
- Enabled: ✓
- Data: PostProcessData (Post Process Dat

Overrides
- Stencil: ☐

Compatibility
- Intermediate Texture: Always

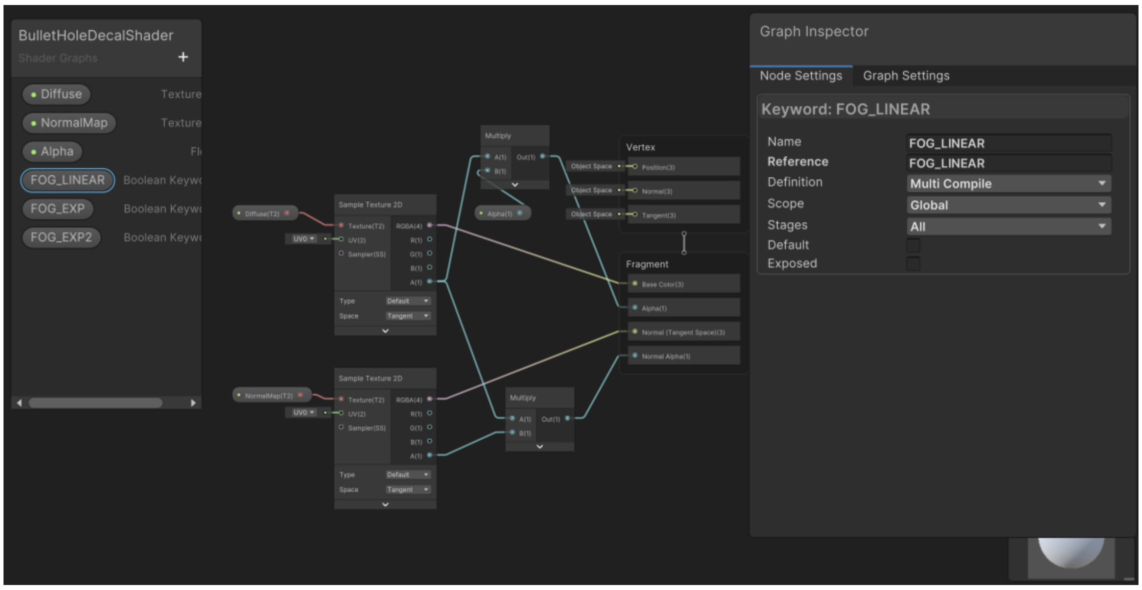
Renderer Features

✓ **Decal**
- Technique: Screen Space
- Normal Blend: Low
- Use GBuffer: ✓
- Max Draw Distance: 1000

✓ **Render Objects (Render Objects)**
- Name: RenderObjects
- Event: BeforeRenderingTransparents
- Filters
  - Queue: Opaque
  - Layer Mask: IgnoreDecals
  - LightMode Tags: 0
- Overrides

Picture 9: Set up of the Forward Renderer settings. IgnoreDecals layer is excluded from Opaque Layer Mask, shown as “Mixed…”. Under Decal Renderer Feature, there’s a new Render Objects feature, set to the event BeforeRenderingTransparents. This feature only includes the IgnoreDecals layer in its layer mask.

Another issue with the bullet holes was encountered when running the game in a standalone Windows build. The game’s fog would colour all the decals in a manner where they would appear in unnatural colour when viewed from a distance. This was fixed in the Decal Renderer shader by adding certain shader Boolean keywords that exclude fog being rendered with the shader and setting their respective values to false. Said shader keywords are FOG_LINEAR for linear fog setup, FOG_EXP for exponential fog and FOG_EXP2 for squared exponential fog. Apart from that the bullet hole decal shader is a very basic shader with albedo colour and normal map properties being supported. Currently the game supports different bullet holes and impact effects for generic, wood, metal, rock, and brick materials. Picture 10 below shows, how this is implemented in the Unity’s Shader Graph editor.



BulletHoleDecalShader
Shader Graphs

Diffuse	Texture
NormalMap	Texture
Alpha	Fl
FOG_LINEAR	Boolean Keyw
FOG_EXP	Boolean Keyw
FOG_EXP2	Boolean Keyw

Graph Inspector

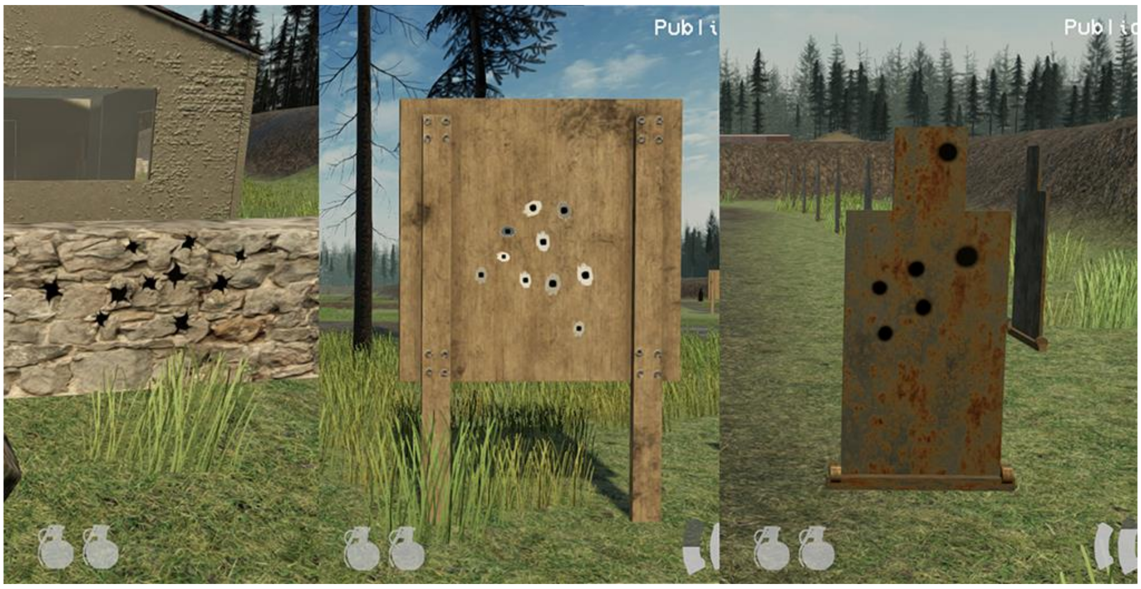
Node Settings | Graph Settings

Keyword: FOG_LINEAR

Name	FOG_LINEAR
Reference	FOG_LINEAR
Definition	Multi Compile
Scope	Global
Stages	All
Default	☐
Exposed	☐

Picture 10: Bullet hole projector shader in Unity’s Shader Graph editor. Keyword set up for the selected keyword (FOG_LINEAR) can be seen in the Graph Inspector window.

Finally, on the next page is the result for how the bullet holes are implemented in the game using the solution above (Picture 11).



Picture 11: Screenshots from the game of the current implementation of bullet holes. Shown from left to right different bullet holes on rock, wood, and metal surfaces. Bullet holes are scaled and rotated randomly, and they affect both albedo colour and normal map properties of the projected surface.

This implementation of bullet holes satisfies the requirements defined in the earlier chapter and it also provided a good base to work on implementing blood splatters and blood pools. It is also easy to extend the system to include variations of bullet holes and adding new materials for new bullet hole and effect types.

4.2 Blood Splatters and Blood Pools

Blood splatters and blood pools follow closely the same implementation as the bullet holes. As these are only to be rendered in the game environment, the static objects, the solution is for the most part based on the same Decal Renderer implementation. A bullet hit on a character is based on the same shot Raycast as before. If the hit is detected with a character collider, we store the raycast hit properties for the blood splatter effect. First another ray is casted from the hit position along the previous shot ray direction. This ray has a predetermined length of for example two units or meters varied slightly for each shot in Unity and the purpose is to check whether there’s an environmental collider right behind the character such as a wall, rock, or a tree. If a hit is detected, the position and surface normal direction of that are stored and used for the blood splatter’s position and orientation.

In case no surface behind the hit is detected, another ray is casted straight down from the endpoint of the previous ray. This is to simulate the blood splatter falling to the ground if there’s no surface at the direction of the hit. As before, if the Raycast detects a hit, its position and surface normal direction are used for positioning and orienting the splatter effect.

Finally, the blood splatter Decal Renderer prefab is instantiated from the pool, just like the bullet holes as described above. A random rotation and scale are applied to the transform of the renderer and a random texture is picked from a selection to avoid repetition.

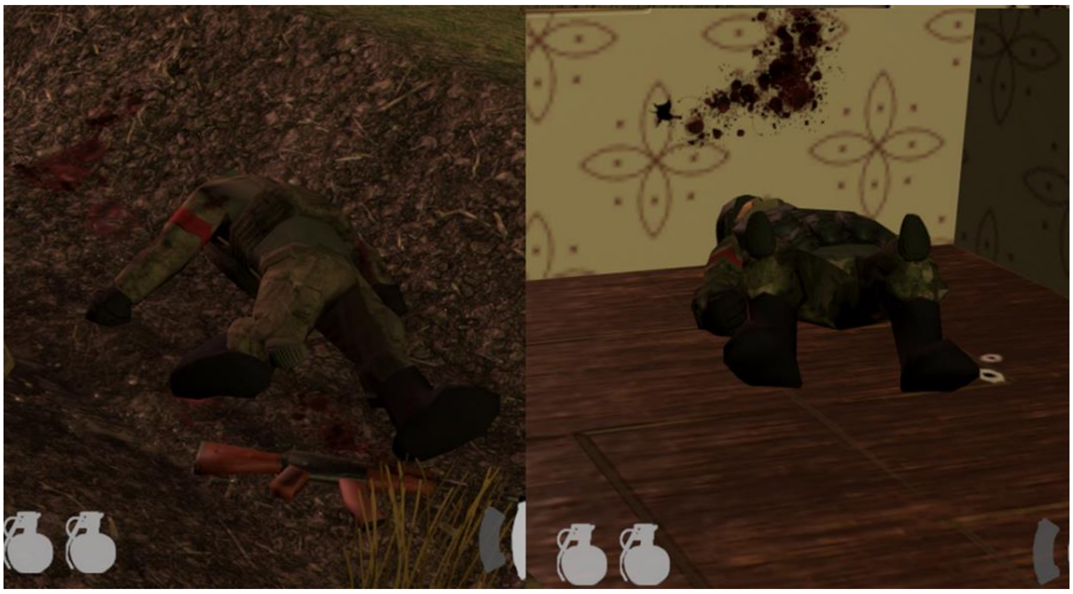
Spawning blood happens in the same class and function as other shooting related effects, which is ShootingEffects::SpawnHitEffects():

```
// Variables for the blood splatter raycast are initiated.
RaycastHit hit;
// Ray origin is offset from the hit position so that it
doesn’t collide with the character itself.
Vector3 rayOrigin = position + shootDirection
Ray ray = new Ray(rayOrigin, shootDirection);
GameObject bloodObject = null;
// Ray is casted with the length of 2.
bool spawnBlood;
if (Physics.Raycast(ray, out hit, 2f,
AssetHolder.Instance.ShootMask))
{
    spawnBlood = true;
```

```
}
else
{
    // First ray didn’t hit so another is casted straight
down to see if splatter can be spawned on the ground
instead.
    rayOrigin = position + shootDirection * 2f;
    ray.origin = rayOrigin;
    ray.direction = Vector3.down;
    if (Physics.Raycast(ray, out hit, Mathf.Infinity,
AssetHolder.Instance.ShootMask))
    {
        spawnBlood = true;
    }
}
if (spawnBlood = true)
{
    // Blood splatter projector is instantiated from the
pool, positioned and rotated.
    bloodObject =
PoolController.Instance.PooledInstantiate(AssetHolder.Insta
nce.bloodDecalProjector);
    bloodObject.transform.SetPositionAndRotation(hit.point,
Quaternion.LookRotation(-hit.normal));
    DecalProjector projector =
bloodObject.GetComponent<DecalProjector>();
// A blood splatter material is randomized for the
projector.
    projector.material =
AssetHolder.Instance.bloodDecalMaterials[Random.Range(0,
AssetHolder.Instance.bloodDecalMaterials.Length)];
```

```
// After that, the projector is scaled and rotated using
the same logic as the bullet hole above. Omitted from the
sample for brevity reasons.
}
```

The picture below (Picture 12) shows how the implementation of blood splatters using the solution described works in the game.



Picture 12: Final implementation of the blood splatters in game. On the left the blood splatters have fallen to the ground and on the right, they are projected on the wall behind the character. Both characters were shot from the camera’s point of view.

Blood pools also follow a very similar implementation utilizing the Decal Renderer feature. When a character dies as a result of an explosion, it is stored in the memory as a Boolean variable. After death, the character’s ragdoll is activated, and its rigidbodies’ velocities are checked every few seconds. When all rigidbody velocities are below a certain threshold value, the ragdoll is frozen in place by setting its rigidbodies kinematic and disabling their respective colliders.

Ragdoll’s state is monitored and frozen in AnimationController class in an IEnumerator function that is run after the ragdoll physics have been activated:

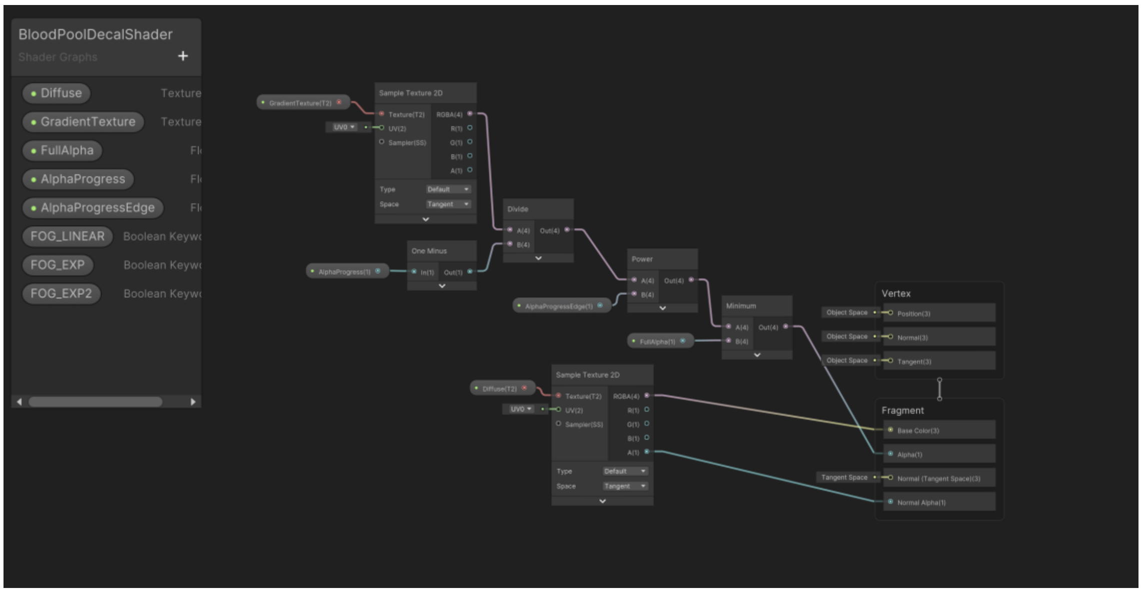
```
private IEnumerator FreezeRagdollRoutine()
{
    bool ragdollFrozen = false;
    while (!ragdollFrozen)
    {
// We iterate this block every 2 seconds until the ragdoll
is frozen.
        yield return new WaitForSeconds(2f);
// ragdollParts is an array in which all the ragdoll bones’
DamageComponent script component is stored.
        foreach (var item in ragdollParts)
        {
            if (item.GetRagdollActive())
            {
// If any of the ragdoll parts are still moving, break
operation.
                freezeRagdoll = false;
                break;
            }
        }
        if (freezeRagdoll)
        {
            foreach (var item in ragdollParts)
            {
                item.ToggleRagdollPart(false);
            }
            ragdollFrozen = true;
        }
// hips is a Rigidbody component variable that is connected
to the character rig’s hip bone.
        if (hips.velocity.magnitude < 1f)
// In this function the blood pool is spawned from the pool
and positioned at the character’s hips position.
Implementation is mostly the same as it is with the other
Decal Projectors.
            SpawnBloodPool();
    }
}
```

DamageComponent class, which is referred to in the previous code sample, handles character related functionality with shooting such as damage multipliers for different body parts. These are the parts that are referred in previous code:

```
// This is called in each DamageComponent when the ragdoll
is frozen or activated.
public void ToggleRagdollPart(bool toggle)
{
    if (!dismembered)
        collider.enabled = toggle;
    rb.isKinematic = !toggle;
}

// This function checks whether this ragdoll part is moving
public bool GetRagdollActive()
{
// rb is a reference to the Rigidbody assigned to this
bodypart and DamageComponent.
    return rb.velocity.magnitude > 0.1f;
}
```

At this time, the blood pool is instantiated from a pool. The character’s position is used as the position for the transform and the rotation is default as the blood pool should always be facing towards the ground below it. Blood pools have a special shader and script that controls the shader. This is to achieve the effect of the blood spreading on the ground around the character over time. For this effect a gradient texture is used to control the alpha property of the shader.

The picture on the next page (Picture 13) shows how all of that is implemented using Unity’s Shader Graph editor.

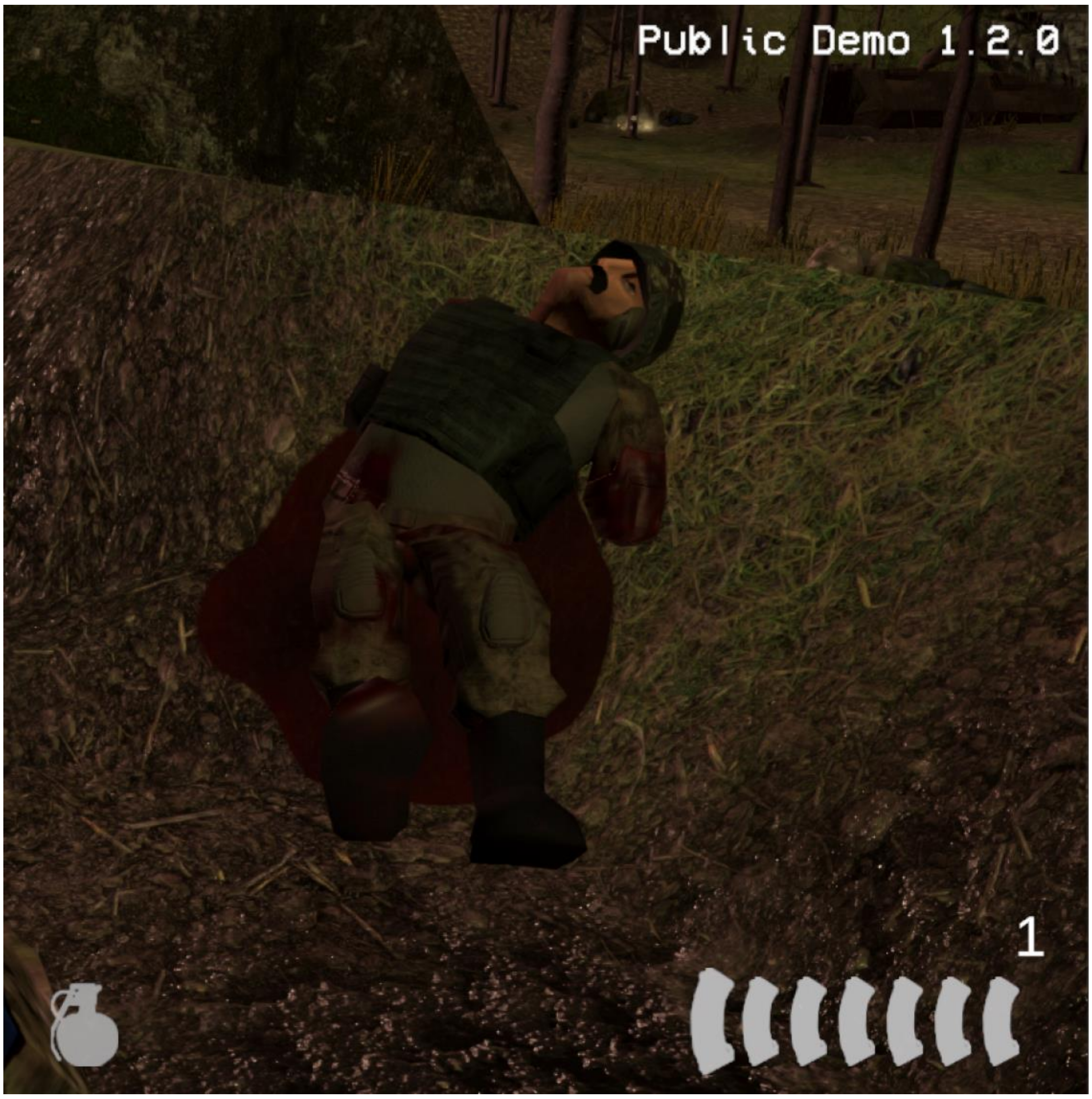


Picture 13: Blood pool shader in Shader Graph. Both Alpha FullAlpha and AlphaProgress values are increased over time via code. Values of the gradient texture are divided with the inverse of the current value of AlphaProgress, which causes the effect of the pool growing over time.

This is how the spreading of blood is controlled via a script component called BloodPoolController attached to the blood pool projector object:

```
using UnityEngine;
using UnityEngine.Rendering.Universal;
public class BloodPoolController : MonoBehaviour
{
    [SerializeField]
    private DecalProjector projector;

    private float spawnTime;
    private float alphaTarget;

    private void Start()
    {
// Projector material is instantiated by picking a random
blood pool texture. The size and final alpha target are
varied randomly.
        projector.material =
Instantiate(AssetHolder.Instance.bloodPoolMaterials[Random.
Range(0, AssetHolder.Instance.bloodPoolMaterials.Length)]);
        projector.size = Vector3.one * Random.Range(0.9f,
1.3f);
        alphaTarget = Random.Range(0.9f, 0.99f);
    }

// This function is run every time the blood pool is
instantiated from the object pool.
    private void OnEnable()
    {
        spawnTime = Time.time;
    }

// _AlphaProgress and _FullAlpha values of the instantiated
material's shader are lerped over time (1 second and 10
seconds respectively).
    private void Update()
    {
        float timeSinceSpawn = Time.time - spawnTime;
        float progressTime = timeSinceSpawn / 10f;
        float alphaTime =  timeSinceSpawn / 1f;
        if (alphaTime < 1f)
            projector.material.SetFloat("_FullAlpha",
Mathf.Lerp(0f, alphaTarget, alphaTime));
        if (progressTime < 1f)
            projector.material.SetFloat("_AlphaProgress",
Mathf.Lerp(0f, 0.9f, progressTime));
    }
}
```

For optimization purposes both blood pools and splatters utilize the same pooling feature as the bullet holes. Instead of destroying old Decal Renderers, they are disabled, hidden, and moved to under a pool transform from which new decals can be instantiated as needed. Furthermore, the players can adjust the upper limits of the amounts for both active blood pools and blood splatters in the game's settings menu. In Picture 14 below, the implementation of blood pools is shown in-game.



Picture 14: Blood pool as it’s implemented in the game. A character has died as a result of grenade explosion and a blood pool has formed under the character.

Both blood splatters and blood pools were implemented as per the specifications defined earlier. Both solutions also leave room for additional variations for blood pool and blood splatter graphics.

4.3 Impact Wounds on Characters

This feature required a more specific implementation since as described previously, Decal Renderers were not an option for skinned and animated characters. It was first tried to tackle the issue by using Unity’s built in

RaycastHit.textureCoord property from the shot Raycast hit. The hit mesh's UV texture coordinates at the collision location are stored in this property and it would thus sound like a reasonable starting point. (19)

It was possible to manipulate the hit mesh's assigned material to draw a single pixel in the affected area on a static object's shader's albedo texture, but it came with some immediate challenges. For a proper implementation using this method, a brush for painting the impact wound around the texcCoord UV coordinates would be required. However, this approach turned out to be complicated and came up with further issues. For example, when the damage would stretch outside a UV edge, there would be a jarring cut off at the edge of the UV island.

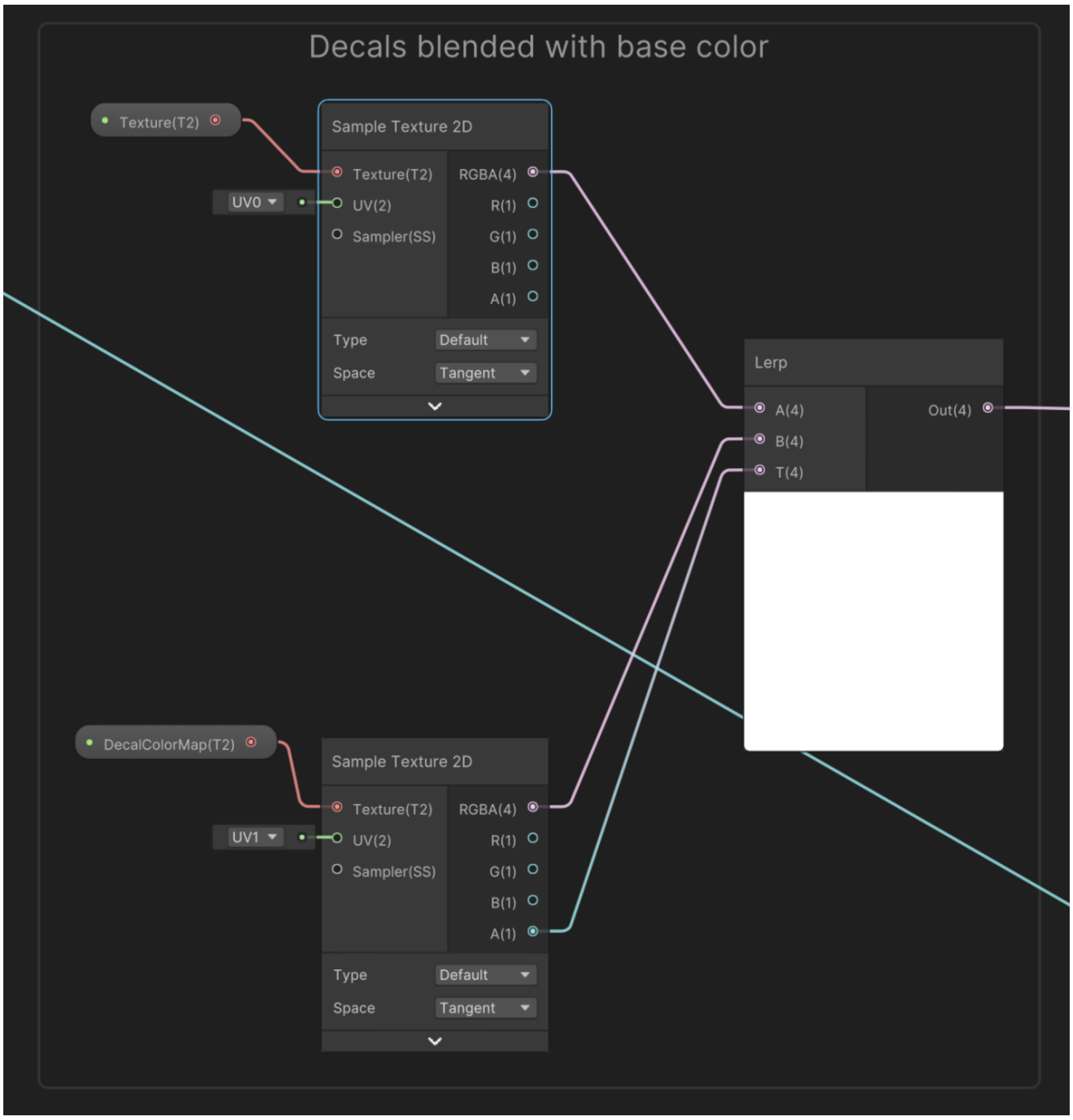
Another issue, even more fundamental of nature, is that the texCoord variable only stores the UV coordinates for a mesh that is directly associated with the hit collider. In more specific terms this means that the mesh and collider must be attached to the same game object and use the same mesh. For my purposes this wouldn't work as the character's mesh is separate from its hitbox colliders, which are primitive collider shapes attached to its bones. Technically this could be solved by baking the character's meshes when a hit is detected, but at this point it was figured that coming up with solutions for all the aforementioned issues was beyond the scope for this feature. It was decided to investigate other approaches.

This is when Skinned Mesh Decals by naelstrof came into picture and it seemed to do exactly what was needed out of the box. In my first iteration, the characters use Unity's primitive shapes such as boxes and capsules as colliders attached to their bones for different body parts. When a hit is detected on any of these colliders, a function to draw a decal using the Skinned Mesh Decals plugin is called with the position of the hit, the affected mesh, size, damage brush material, and rotation of the hit for which the hit surface's normal direction is used. The damage brush material is then used to draw the decal on a Render Texture that is used in the character's body renderer's shader as a detail map, which is drawn on top of the normal albedo texture.

```
// The function in MedicalSystem class is called when a
shot on character is detected. pos and rot variables are
the shot hit point and collision surface normal rotation.
private void DrawBulletHole(Vector3 pos, Quaternion rot)
{
// Random size for the impact wound.
    float s = Random.Range(0.08f, 0.12f);
    SkinnedMeshDecals.PaintDecal.RenderDecal(torsoRenderer,
AssetHolder.Instance.ImpactWoundMaterial, pos, rot,
Vector2.one * s, 100f);
}
```

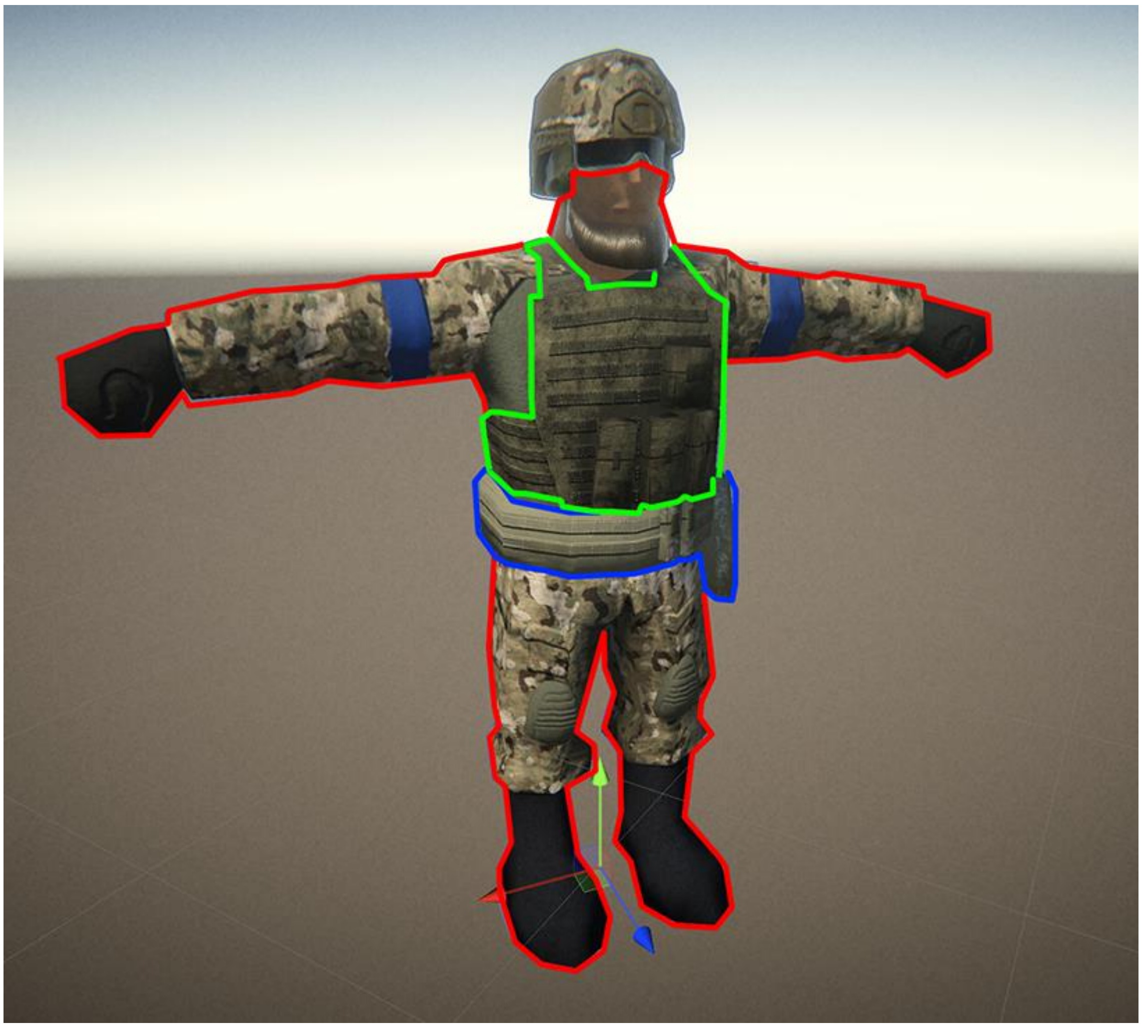
To make the character model assets work with the Skinned Mesh Decals plugin, an additional UV coordinate channel for the required meshes was created in Blender. It could’ve been done automatically in Unity using the lightmap generation feature, but it was figured that there might be a need to use the lightmap UV coordinates later for some other purpose and that making new UV maps in Blender wouldn’t take mora than a few seconds anyway, so it was decided to opt for that to ensure full control over the UV channels.

Picture 15 on the next page shows how the two textures, the character’s main texture properties “Texture” and “DecalColorMap” are combined using a lerp function.



Picture 15: Screenshot of the character shader in Unity Shader Graph editor. This part of the shader blends the characters albedo texture property on UV channel 0 with the DecalColorMap texture on UV channel 1 with a lerp function using DecalColorMap's alpha channel as the lerp value.

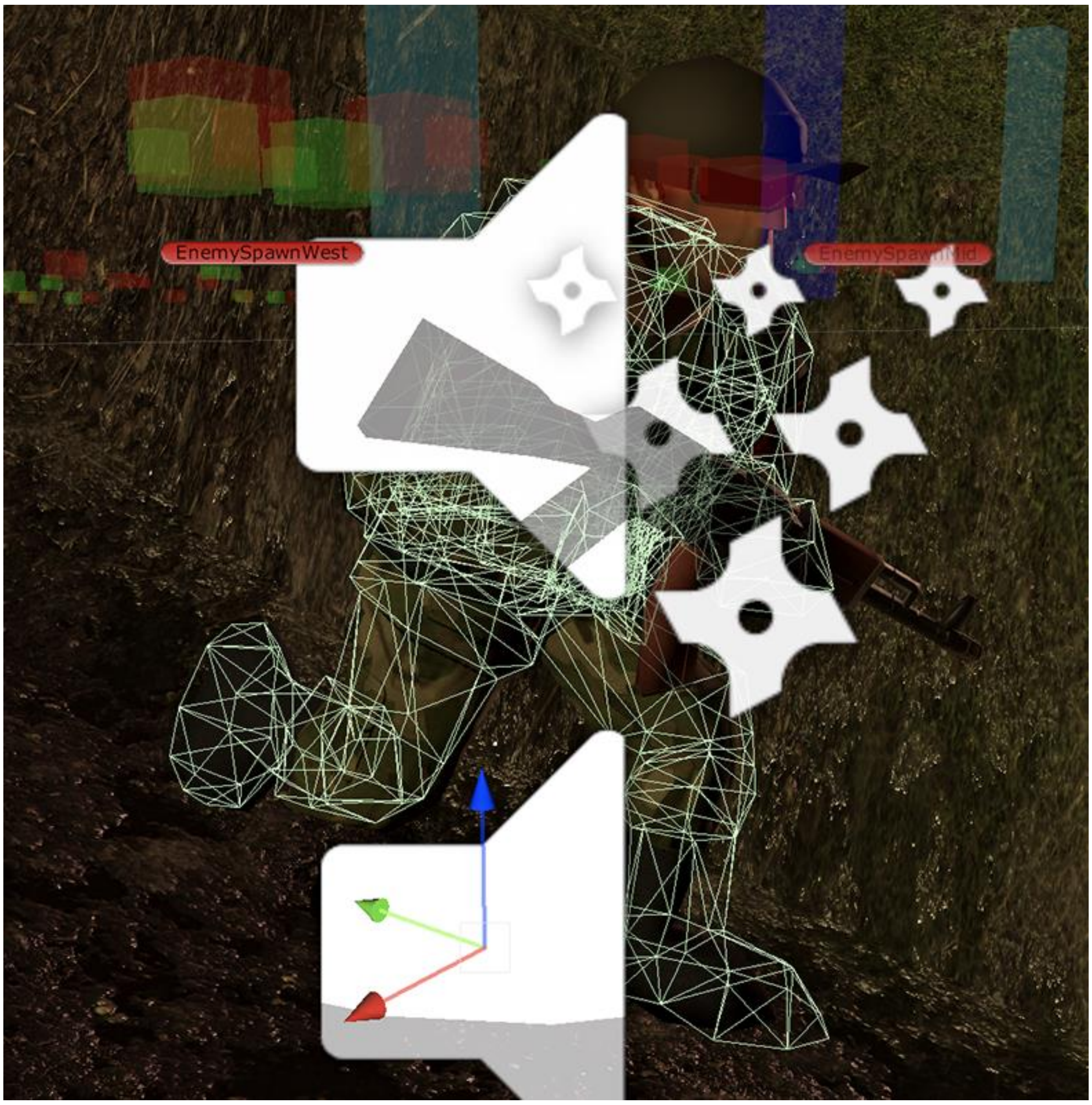
This approach yielded fairly good results and worked as an excellent proof of concept; certain issues were noticed with it that needed to be fixed. First, the characters have multiple different meshes to draw impact wounds to. Players can customize their character's look with different material and model options and for non-player-characters said options are randomized, so there are few different meshes that go with each character. In this context, there are three meshes that are relevant: the character's body mesh, the gear mesh that is for load bearing equipment – chest rig, plate carrier or similar, and belt mesh which is an optional belt for the character. In the first iteration impact wounds were only drawn on the body mesh. This issue will become noticeable if the shot hits the character on their gear mesh, for example and no impact wound would show as it would be drawn on the surface of the body mesh under the gear mesh. The picture below (Picture 16) shows how the different meshes combine to form the game character.


Picture 16: Game character's meshes highlighted in red for body, green for gear and blue for belt.

Another issue with the first iteration was that while the primitive colliders give a good enough estimation of the character's pose for technical gameplay purposes for the most part, they are just primitive shapes parented to the character rig’s bones. This means that the colliders only follow the character’s pose roughly since they do not take the skinning of the mesh into account. Sometimes this would cause the impact wound to be slightly off from where the hit took place or not appear at all.

The solution for these issues in the final version is to use Unity’s BakeMesh function for SkinnedMeshRenderer component. This function creates a snapshot of the skinned mesh of the renderer in its current pose and stores it in a Mesh variable that can be used for multiple purposes. (20) BakeMesh In this case, if a hit is detected on a character, its body, gear, and belt mesh are baked using the function and used as colliders. A new Raycast using the shot ray’s origin and direction is casted and the game check’s where on these baked meshes the shot collides at. This gives a much more accurate position for the impact wound decal to be drawn on each individual mesh.

Picture 17 on the next page is a screenshot from Unity Editor showing a collider mesh that has been baked from the skinned and animated character on top of the character.

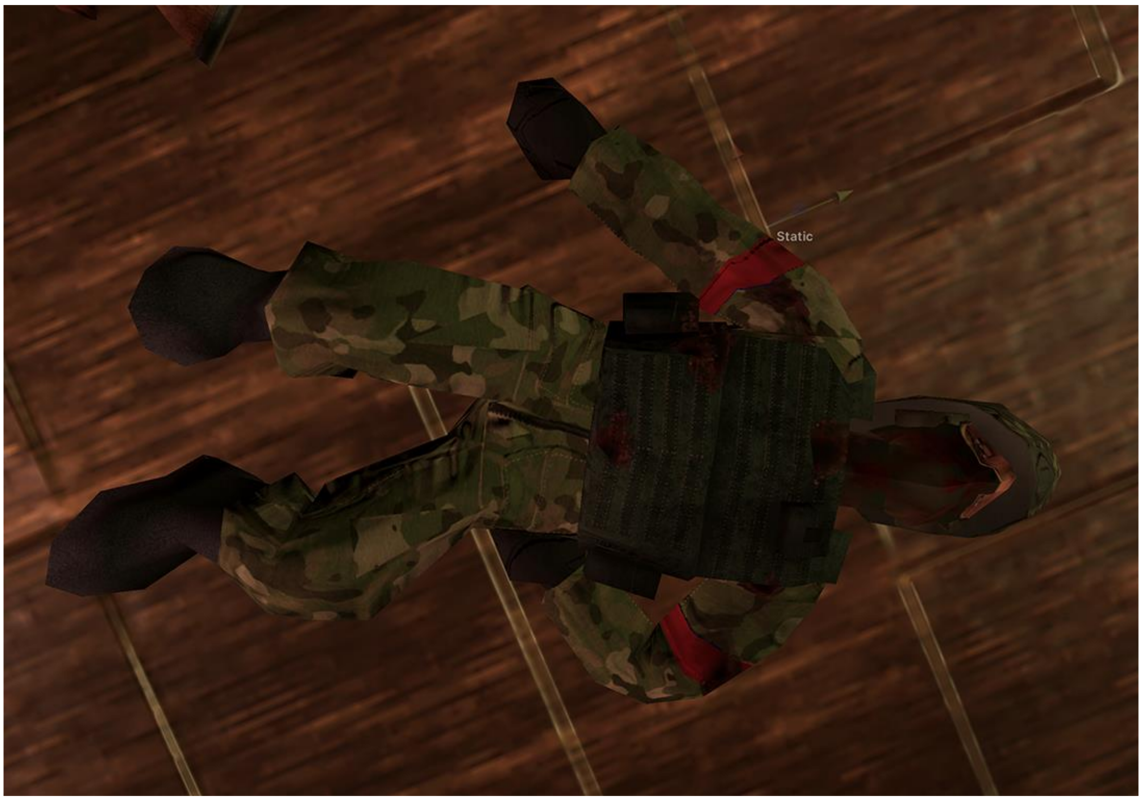


Picture 17: Picture of the baked torso, belt and gear mesh colliders shown as green meshes around the character.

```
private void DrawBulletWound(Vector3 origin, Vector3
destination)
{
    bool beltActive = beltRenderer.gameObject.activeSelf;
    Quaternion rot = Quaternion.identity;
    Vector3 pos = Vector3.zero;
// This extension method makes a normalized directional
vector between origin and destination points.
```

```
    Vector3 direction = Extensions.Direction(origin,
destination);
// Activate the body, belt and gear mesh collider objects.
    torsoCollider.gameObject.SetActive(true);
    if (beltActive)
        beltCollider.gameObject.SetActive(true);
    gearCollider.gameObject.SetActive(true);
    float s = Random.Range(0.08f, 0.12f);

    RaycastHit hit;
    Ray ray = new Ray(origin, direction);

// Mesh colliders are baked and assigned
    Mesh bakedTorso = new Mesh();
    torsoRenderer.BakeMesh(bakedTorso);
    torsoCollider.sharedMesh = bakedTorso;

    Mesh bakedGear = new Mesh();
    gearRenderer.BakeMesh(bakedGear);
    gearCollider.sharedMesh = bakedGear;

    if (beltActive)
    {
        Mesh bakedBelt = new Mesh();
        beltRenderer.BakeMesh(bakedBelt);
        beltCollider.sharedMesh = bakedBelt;
    }

// Ray is casted to the mesh colliders and if hit is
detected, it is used for the decal position and rotation.
    if (Physics.Raycast(ray, out hit, Mathf.infinity,
AssetHolder.Instance.CharacterGraphicsMask))
    {
        pos = hit.point;
        rot = Quaternion.Euler(hit.normal);
    }
    if (rot != Quaternion.identity && pos != Vector3.zero)
    {
SkinnedMeshDecals.PaintDecal.RenderDecal(torsoRenderer,
AssetHolder.Instance.ImpactWoundMaterial, pos, rot,
Vector2.one * s, 100f);
SkinnedMeshDecals.PaintDecal.RenderDecal(gearRenderer,
AssetHolder.Instance.ImpactWoundMaterial, pos, rot,
Vector2.one * s, 100f);
if (beltActive)
    SkinnedMeshDecals.PaintDecal.RenderDecal(beltRenderer,
AssetHolder.Instance.ImpactWoundMaterial, pos, rot,
Vector2.one * s, 100f);
    }
// After the operation, mesh colliders are hidden again.
    torsoCollider.gameObject.SetActive(false);
    gearCollider.gameObject.SetActive(false);
    if (beltActive)
        beltCollider.gameObject.SetActive(false);
}
```

The game’s final implementation for impact wounds, using the solution described above is shown on the next page in Picture 18.



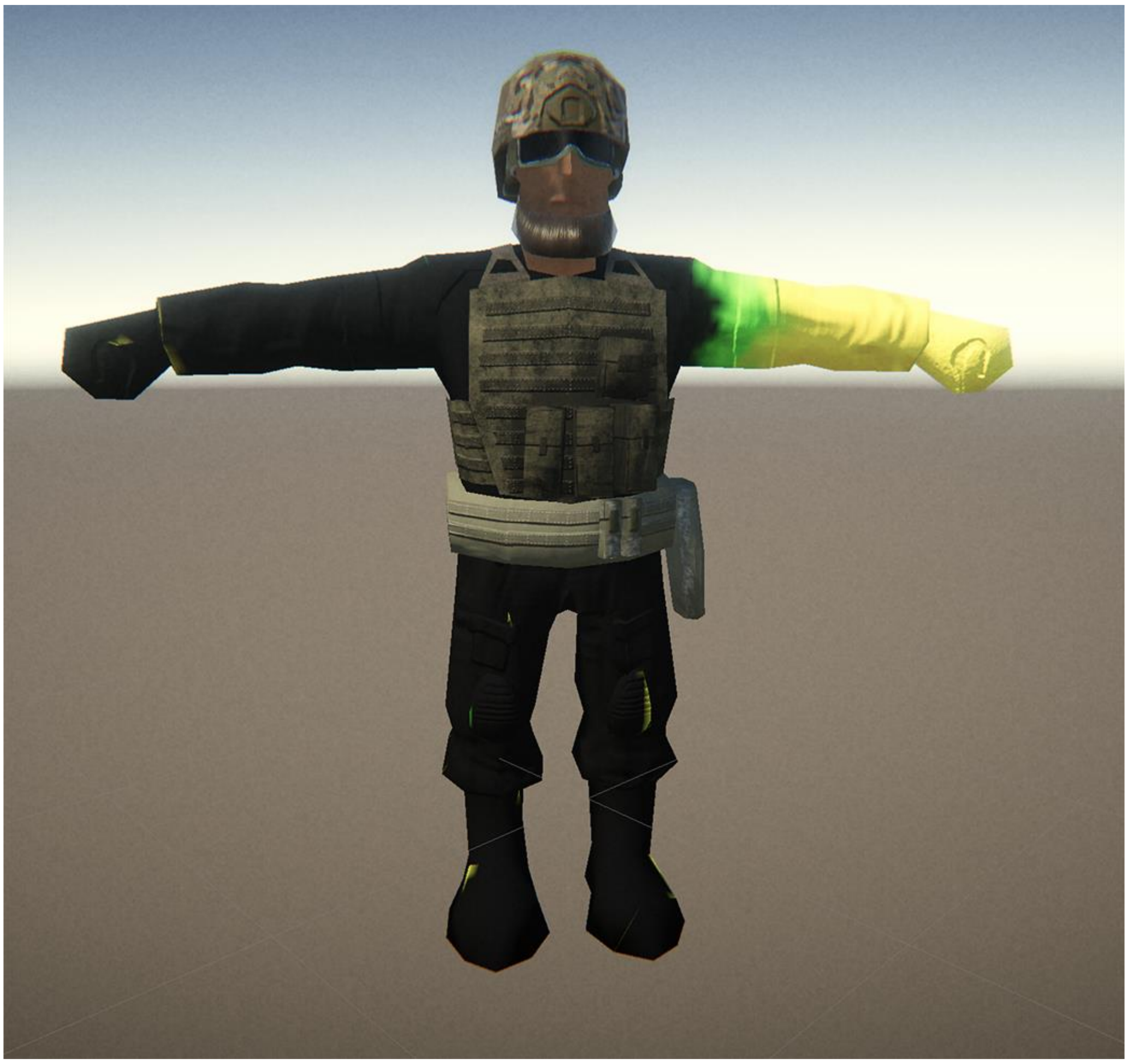
Picture 18: Current implementation of the impact wounds. Character has been shot from the front to his upper torso, arms, and head.

For the most part, this implementation works well and satisfies the requirements that were set earlier. There are slight issues, that appear occasionally where the impact wounds appear stretched when hitting in a shallow angle. Additionally, sometimes the impact wounds appear in varying sizes between the different meshes. Regardless, the implementation of the feature proves good enough for this stage of the production and said issues can be addressed later in the game’s production timeline.

4.4 Generic Blood Overlay

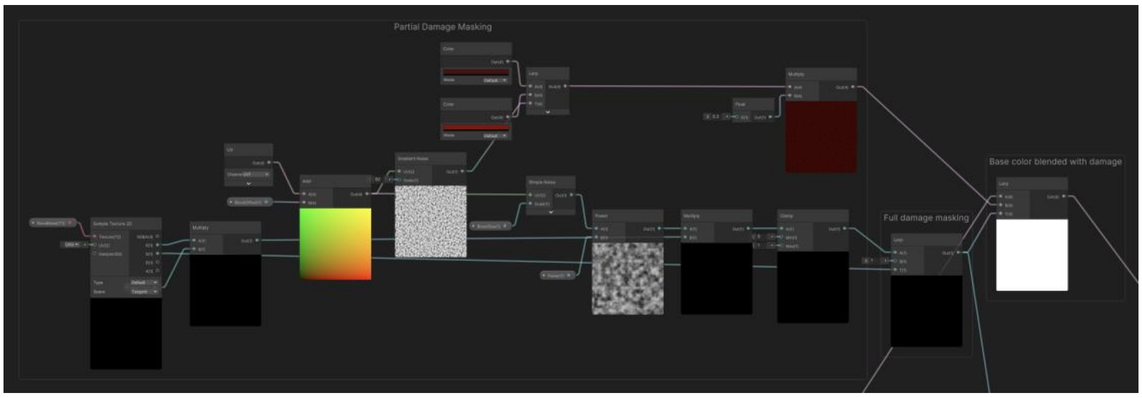
The final requirement for the response mechanics that involve modifying the textures and materials of objects during the game’s runtime is the bloody overlay for explosion deaths. As described earlier, Decal Renderers could technically get the job done, but are not the most efficient solution for purposes which involve animated characters.

For the implementation a script is attached to the collider primitives that are each connected to a different body part for the characters' health system. This is used also for gameplay purposes – a shot to a leg does a different amount of damage than a shot to the head for example, but for the intents and purposes of the explosion damage overlay system this variable is used for visualizing damage on different body parts. Each body part of the character has a an enum value for its corresponding body part in its DamageComponent component. The medical system has a damage 128x128 texture assigned for each of these body parts. The texture assets were created using Substance Painter. A damage mask texture is for the character's lower left arm is shown in the picture below (Picture 19).



Picture 19: Damage mask texture for left lower arm as it is currently implemented, replacing the albedo texture of a character. Green colour is used for masking partial damage while red colour is used for damage that covers the whole area. Combination of red and green sums up to yellow.

The character’s shader has a bloody overlay which is based on two colours between which a lerp function is used with a generated simple noise function. For each character the offset of the noise is randomized which adds variations to the overlay avoiding repetition. The blood overlay also affects the shader’s smoothness value to achieve a wet and blood-soaked appearance on the affected areas. The damage overlay is masked on the character using a lerp function node with a DamageMask named texture property which is by default a 128x128 black texture. The green channel of the texture masks full damage and red channel is for partial damage. Below is a screenshot (Picture 20) of said implementation in Unity’s Shader Graph editor.

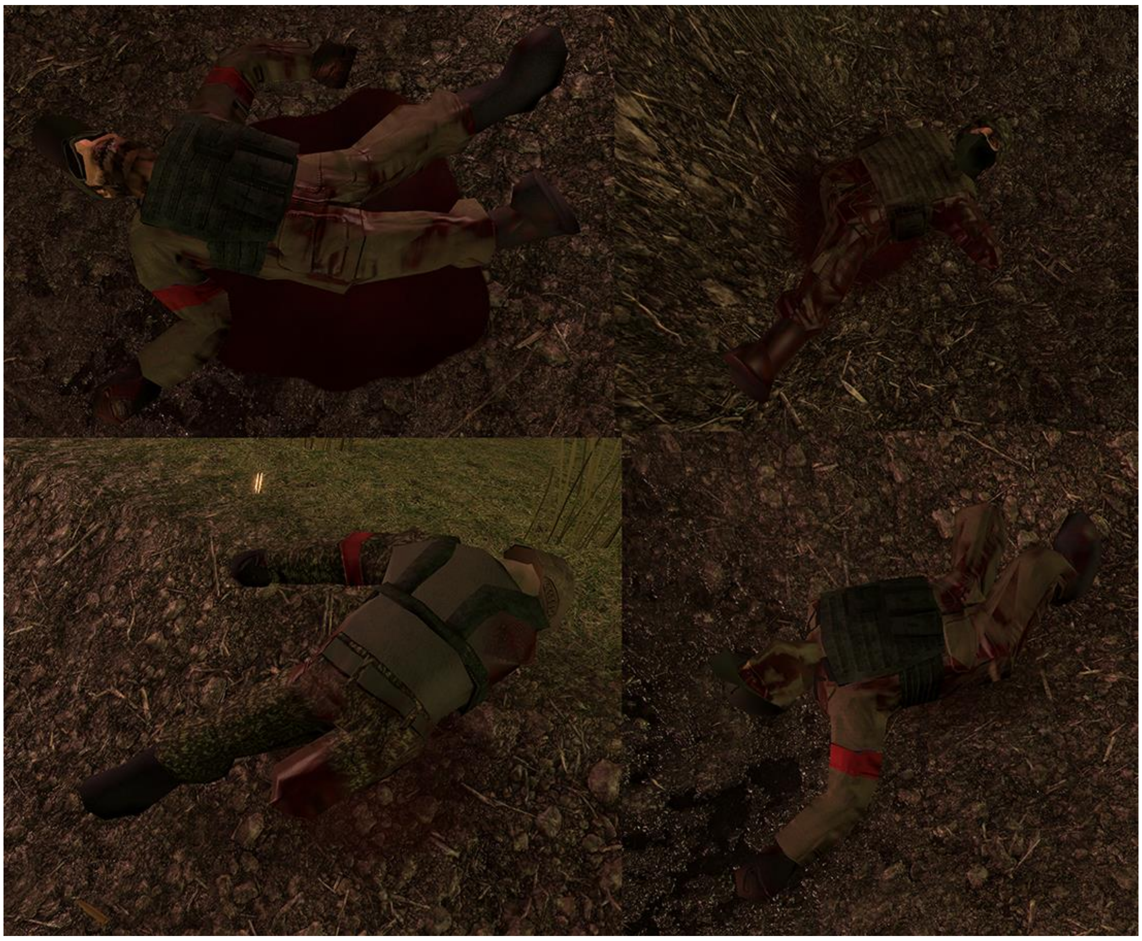


Picture 20: Damage Masking in character shader shown in Unity’s Shader Graph editor. The damage itself is just a lerp between two shades of red colours using a procedural noise texture. Partial damage UV layout is randomly offset using BloodOffset property. Partial damage is masked using BloodMask texture property’s red channel. Full damage is masked by using the green channel.

When a character takes damage and the appropriate damage mask texture for the damaged body part is selected as described above, the characters material is instantiated. The associated DamageMask texture is instantiated as well. The instantiaton is done so that we can modify an instance of the material and the texture – not the material and texture for all the characters sharing the shader. After that the values of each pixel of the hit body part's damage mask texture are stored to an array which is then applied to the shader's DamageMask texture. This way we can add multiple damaged parts to the DamageMask texture which allows us to render the bloody overlay on multiple damaged body parts simultaneously.

```
// tex is the damage mask texture for the part we want to
apply. colorIndex is used for different colors for
different damage masks (partial or full damage).
private IEnumerator ApplyDamageTextureRoutine(Texture2D
tex, int colorIndex)
{
    yield return new WaitForEndOfFrame();
// Get pixels from the character’s damage mask and the
applied body part’s damage mask.
    Color[] damageMaskPixels =
damageMaskTexture.GetPixels(0);
    Color[] damageTexturePixels = tex.GetPixels(0);
    Color32[] finalColors = new
Color32[damageMaskPixels.Length];
// Iterate through the pixels
    for (int i = 0; i < finalColors.Length; i++)
    {
        Color finalColor = damageMaskPixels[i];
// Check whether we want to add red or green values to the
damage mask and add it to the right channel.
        float colorToAdd = colorIndex == 0 ?
damageTexturePixels[i].r : damageTexturePixels[i].g;
        if (colorIndex == 0)
            finalColor.r += colorToAdd;
        else if (colorIndex == 1)
            finalColor.g += colorToAdd;
        finalColors[i] = finalColor;
    }
// Apply pixel color values from the finalColors array to
the damage mask texture.
    damageMaskTexture.SetPixels32(finalColors, 0);
    damageMaskTexture.Apply();
}
```

Below is a screenshot from the game (Picture 21) showing how the generic bloody overlay works in the game using the solution described above.


Picture 21: Varying examples of generic bloody overlay as it’s implemented to the game.

Blood overlay in its current implementation satisfies the requirements set for the feature well and making damage masks for future body meshes is a quick process. This makes the feature easily scalable for future content production.

5 Conclusion

5.1 Results and Analysis

As so often is the case with technical art related challenges, there are multiple ways to achieve the same goals and end results, especially if the use cases are varying even in minor ways. Often there are multiple solutions to same problem even with only one, specific use case, and it is often hard or impossible to quantify which solution is the best, most efficient, and most optimal. This is because there are so many ways different metrics and measures to consider.

All the above apply to the different use cases of painting on 3D objects and characters in Unity engine very well. Even though the use cases are quite similar on the surface, the solutions and technologies utilized vary considerably. The clearest difference between the use cases and the final solutions for them was whether we are working with static meshes or skinned and animated characters. This and other differences between the use cases would directly limit the use of certain technologies and solutions or indirectly make one or the other solution be more efficient or better achieve the wanted result.

The solutions that were ended up with during this study work well. In all use cases the predefined end results were achieved and none of the solutions are too heavily affecting the game’s performance at current use rates. Furthermore, the implementations are generic enough so that they can be repurposed for either other use cases in the same or in future game projects, which would speed up significantly the development of future games when applicable use cases appear. Developing such features in future projects will directly improve the game feel and responsivity.

5.2 Future Considerations

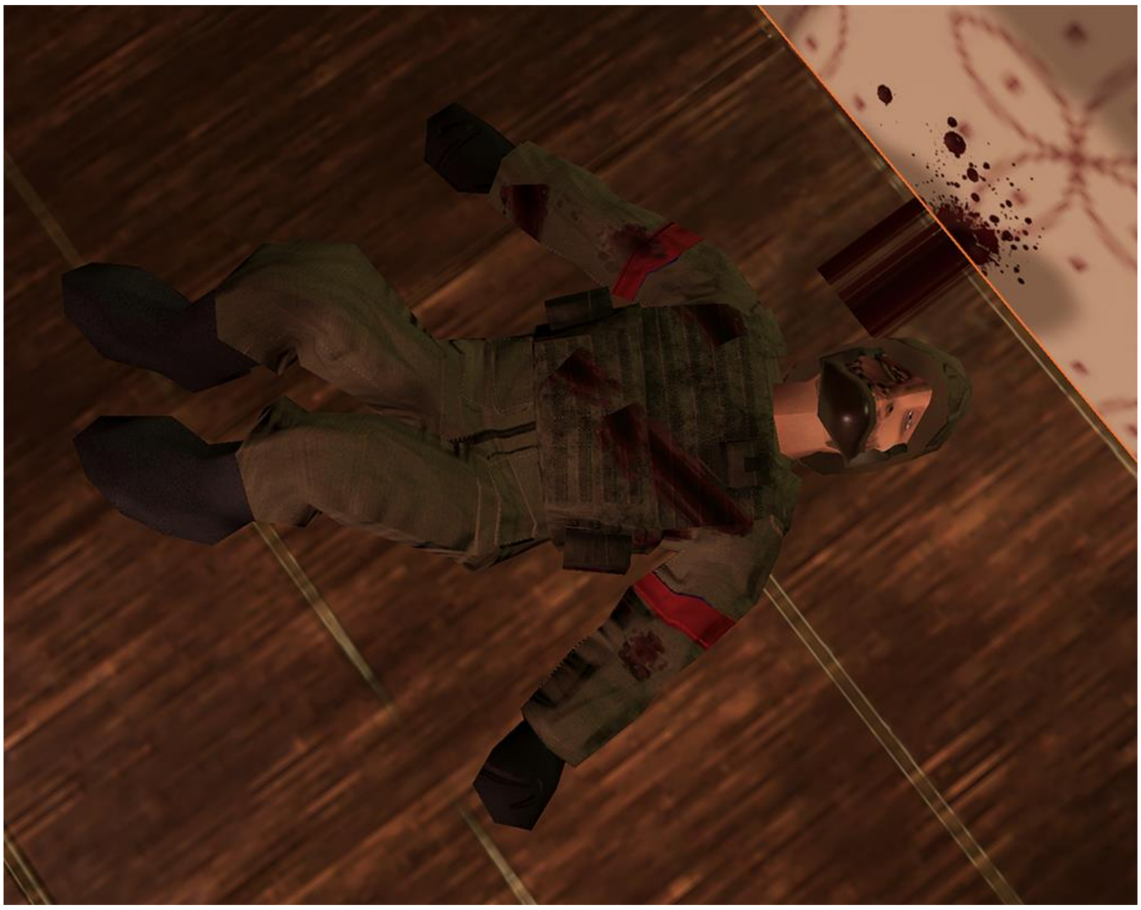
While the current iteration of implementations is sufficient for the time being, there is clearly lots of room for further improvement as well. Monitoring and improving the game’s performance is something that needs to be constantly investigated holistically, and these solutions are not exempt to that rule. Especially the use and manipulation of texture assets and render textures in runtime and using instances of materials and textures is something that will have an impact on the game’s memory usage. On the other hand, rendering Decal Renderers, while considerably light, is still limited to some extent by the CPU capacity.

Unity’s built-in tool for runtime performance monitoring, Profiler, is very handy for these purposes. Unity’s profiler can show the runtime loads for both memory and processor usage, and furthermore, which scripts or assets are taking the most capacity (21). Thorough analysis the profiler output also gives good ideas, which parts of the game should be considered first when further optimizations and other improvements take place in later parts of the development.

One potential way of improving the runtime performance cost of the damage overlay would be to investigate whether rendering the damage mask texture onto a Render Texture asset would be feasible. For this purpose, a similar shader-based solution as with the Skinned Mesh Decals would be required. Such shader would first fit the damage mask texture on screen, then draw the added damage onto it and then render the result on a Render Texture which could then be used in the character’s material shader. This would potentially be less performance intensive than changing the colour values of the individual pixels or the texture pixel array.

In addition to runtime performance improvements, the implementations leave room for future aesthetic improvements as well. The most straightforward one would be to add more variety in the textures used for bullet holes, blood splatters, blood pools and impact wound brushes. The systems already support randomizing the decals and brushes from an array of multiple different ones so adding more variety is just a matter of making more art assets. Additionally, as the game expands and there’s more variety in the art assets, eventually more material types for different bullet holes are required. In its current state the game has hit material types for generic, rock, brick, wood, and metal. Additional ones could include different colour variations of the existing materials, or completely new ones such as concrete, mud or ice for example.

The impact wound feature still has some inconsistencies that require further investigation and fixes. The size of the damage brush appears to vary between the different meshes, which hints that the scaling of the brush happens currently in UV space, as opposed to world space where it should happen. Additionally, further research into the functionality of the brush shader would likely yield some useful results. It would be worth exploring whether rotating the brush either randomly or over time could be used to add random rotation to the impact wounds thus adding more variety. Depth check for the shader would be useful as well since sometimes the impact wounds seem to stretch across the mesh when hit in shallow angles. The shader has been written in HLSL and in the Unity editor preview it only shows as error shader, so iterating it has not been easy. Changing the decal function to use a material with another shader seems to yield very inconsistent and wrong looking results as well painting the whole mesh in one colour for example. The issue is shown in the picture on the next page. (Picture 22).



Picture 21: Example of the glitching impact wounds around the arms and middle torso of the character. The character was shot in a shallow angle which may or may not be related to the issue. Albeit not related to the issue at hand, it is notable that the blood splatter Decal Renderer on the wall is also glitching on the floor.

Despite all the described issues and challenges for further development, the features, as currently implemented, work well enough to go into production and as a conclusion, the found solutions appear to form a fine basis for developing for future use cases as well.

References

1 Juice it or lose it – a talk by Martin Johansson. & Petro Purho, 2012 - https://youtu.be/Fy0aCDmgnxg - Accessed January 31st, 2023.

2 Response Mechanics in Games – Random Madness - https://youtu.be/U4-lOPreM0U - Accessed January 31st, 2023.

3 Para Bellum – Frontlines on Itch.io - https://laukaus-games.itch.io/para-bellum-frontlines - Accessed January 31st, 2023.

4 Persistent Bullet Hole – GiantBomb https://www.giantbomb.com/persistent-bullet-hole/3015-580/ - Accessed March 2ndt, 2023.

5 Skinned Mesh Decals on GitHub, naelstrof - https://github.com/naelstrof/SkinnedMeshDecal - Accessed January 31st, 2023.

6 Unity (Game Engine) on endoflife.date - https://endoflife.date/unity - Accessed March 2ndt, 2023.

7 Unity 2021 LTS - https://unity.com/releases/2021-lts - Accessed January 31st, 2023.

8 Universal Render Pipeline overview – Unity Manual - https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@15.0/manual/index.html - Accessed January 31st, 2023.

9 Shader Introduction – Unity Manual - https://docs.unity3d.com/Manual/shader-introduction.html - Accessed January 31st, 2023.

10 HLSL in Unity – Unity Manual - https://docs.unity3d.com/Manual/SL-ShaderPrograms.html - Accessed March 2ndt, 2023..

11 Decal Renderer Feature – Unity Documentation https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@12.0/manual/renderer-feature-decal.html - Accessed January 31st, 2023.

12 Unity Shader Graph - https://unity.com/features/shader-graph - Accessed January 31st, 2023.

13 Texture2D.SetPixel – Unity Manual - https://docs.unity3d.com/ScriptReference/Texture2D.SetPixel.html - Accessed March 2ndt, 2023.

14 Texture2D.SetPixels – Unity Manual - https://docs.unity3d.com/ScriptReference/Texture2D.SetPixels.html - Accessed March 2ndt, 2023.

15 Mipmaps introduction – Unity Manual - https://docs.unity3d.com/Manual/texture-mipmaps-introduction.html - Accessed January 31st, 2023.

16 Render Texture – Unity Manual - https://docs.unity3d.com/Manual/class-RenderTexture.html - Accessed January 31st, 2023.

17 Physics.RaycastHit – Unity Manual - https://docs.unity3d.com/ScriptReference/RaycastHit.html - Accessed January 31st, 2023.

18 Introduction to Object Pooling – Unity Learn - https://learn.unity.com/tutorial/introduction-to-object-pooling - Accessed January 31st, 2023.

19 RaycastHit.textureCoord – Unity Documentation - https://docs.unity3d.com/ScriptReference/RaycastHit-textureCoord.html - Accessed January 31st, 2023.

20 SkinnedMeshRenderer.BakeMesh – Unity Documentation - https://docs.unity3d.com/ScriptReference/SkinnedMeshRenderer.BakeMesh.html - Accessed January 31st, 2023.

21 Unity Profiler – Unity Documentation https://docs.unity3d.com/Manual/Profiler.html - Accessed March 2ndt, 2023.