Decalery manual

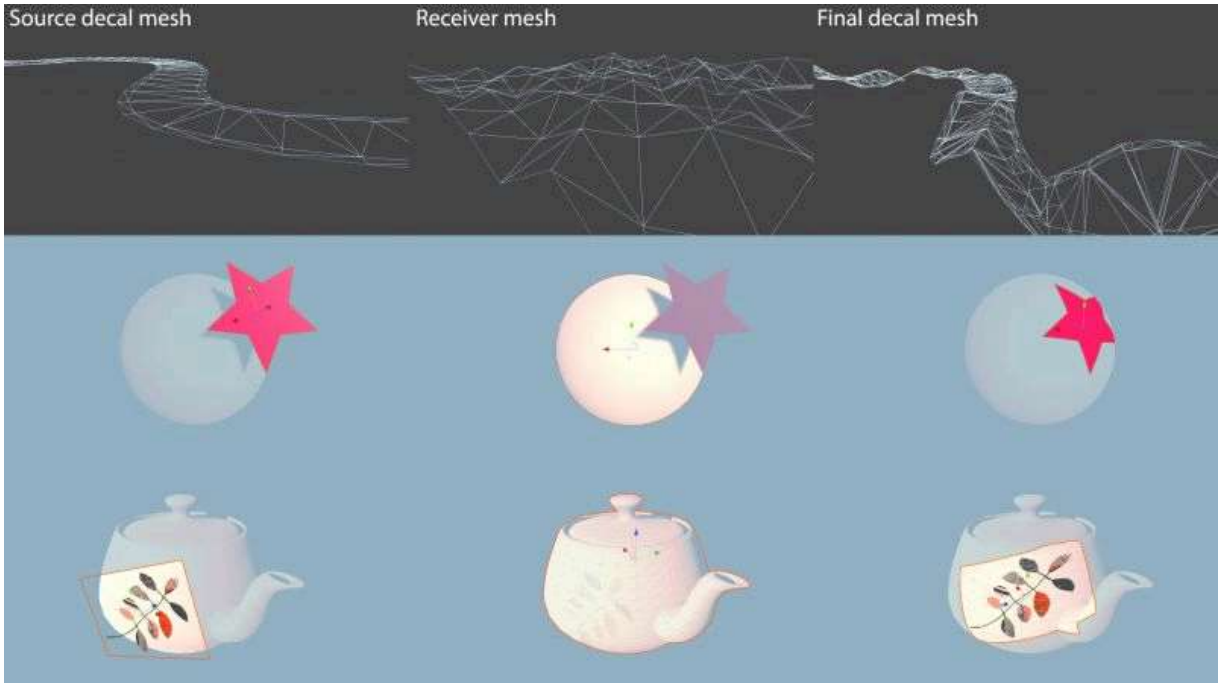
Contents

What is Decalery

In computer graphics, decals are textures projected on other geometry, like stickers, usually overlaid above regular textures to add more detail. There are many ways to implement them.

Decalery is a mesh-based decal system. That is, every decal created with it is a regular mesh made of triangles. This way they can be rendered on any platform and render pipeline, and the only thing the decal shader needs to do is to offset vertices slightly towards the camera (Decalery includes such shaders, and existing shaders can be easily extended).

Decalery projects **source decal mesh** onto a set of **receiver meshes** to generate **final decal mesh**:

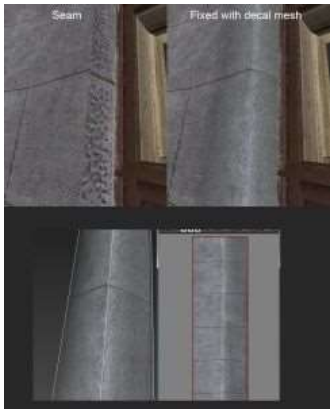


This way Decalery is not limited to simple quad-like decals. For example, mesh decals can be used for:

- Classic quad projections (graffities, bullet holes, paint splats, etc).
- Projected roads/paths on top of curved geometry.
- Edge decals like this (https://simonschreibt.de/gat/fallout-3-edges/).
- As a 3D "clone stamp" to hide texture seams or unwanted tiling.
- etc...

Using Decalery for classic quad decals

Using Decalery for textured paths



Using Decalery for edge trim decals

Using Decalery as a "clone stamp" to paint parts of the wall texture over windows

Decalery supports:

- Decals and receivers of **any geometric shape**.
- Lit and **lightmapped** decals. Decalery supports both built-in and Bakery lightmaps (including SH, MonoSH and Volumes).
- Deformable decals on **skinned** meshes.
- Cross-platform light-weight decal rendering for in-editor placed decals.
- A variety of techniques for runtime decal spawning.
- Trail-like runtime decals for e.g. skidmarks.

When generating final decal meshes, Decalery takes:

- Normals and lightmap UV from the receiver mesh;
- Texture UVs and vertex colors from the source decal mesh.

Decal triangles are perfectly aligned to the receiver mesh, not pushed by normal, so it is impossible to see them "floating". To prevent Z-fighting, a compatible shader is used. Everything such shader does amounts to:

```
worldPos += normalize(_WorldSpaceCameraPos - worldPos) * viewPushConstant // defaults to 0.001
```

On top of that, a little bit of regular slope-scaled depth bias (https://docs.unity3d.com/Manual/SL-Offset.html) is applied. Something like

```
Offset -0.01, -0.01
```

should be enough.

This way decals perfectly replicate underlying surface shading and lightmapping, while keeping their own UVs and vertex color. Vertex color can be used to control opacity, and default Decalery shaders assume that. Per-vertex alpha values take less memory than texture masks most of the time.

Installation

Simply import the package into Unity. Files will be unpacked to **Assets/Decalery**.

There are three example scenes: **example_decals_editor** for in-editor level design usage, **example_decals_runtime** for in-game decal spawning and **example_decals_bakery** showcasing decals sampling Bakery SH, MonoSH and Volumes.

Example scenes are made in Unity 2019 and can be opened with this version or newer; the decal system itself works on older versions. On non-2019 you may need to **rebake lightmaps** in example scenes. example_decals_bakery scene also requires Bakery.

If project uses **URP**, also unpack DecaleryURP.unitypackage; For the Bakery example scene to work, also extract Bakery_ShaderGraphURP.unitypackage that is installed with Bakery.

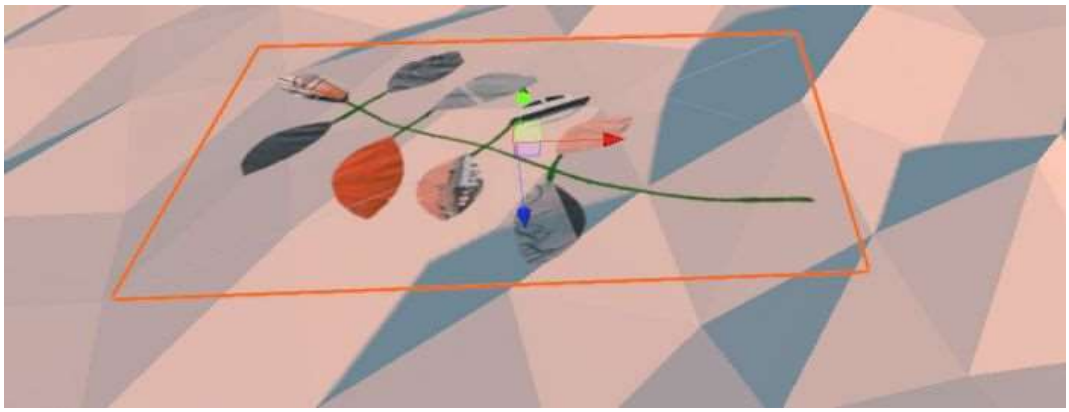
If project uses **HDRP**, unpack DecaleryHDRP.unitypackage. When viewing example scenes, select DefaultSettingsVolumeProfile.asset and in HDRI Sky section, change Exposure Compensation to 1, otherwise scenes will appear too dark. To use Bakery features, also extract Bakery_ShaderGraphHDRP.unitypackage that is installed with Bakery. In HDRP 10 (newer versions are OK) you may need to click on decals to make their material settings appear in Inspector to force HDRP to update their shaders and enable alpha masks.

On Unity 2018 you may encounter "Assembly has reference to non-existent assembly" error. In this case, delete Assets/Decalery/Decalery.asmdef.

Quickstart

Creating decals in Editor

1. Position decal mesh (e.g. a default Quad or anything else) in front of the receiver mesh (or meshes).

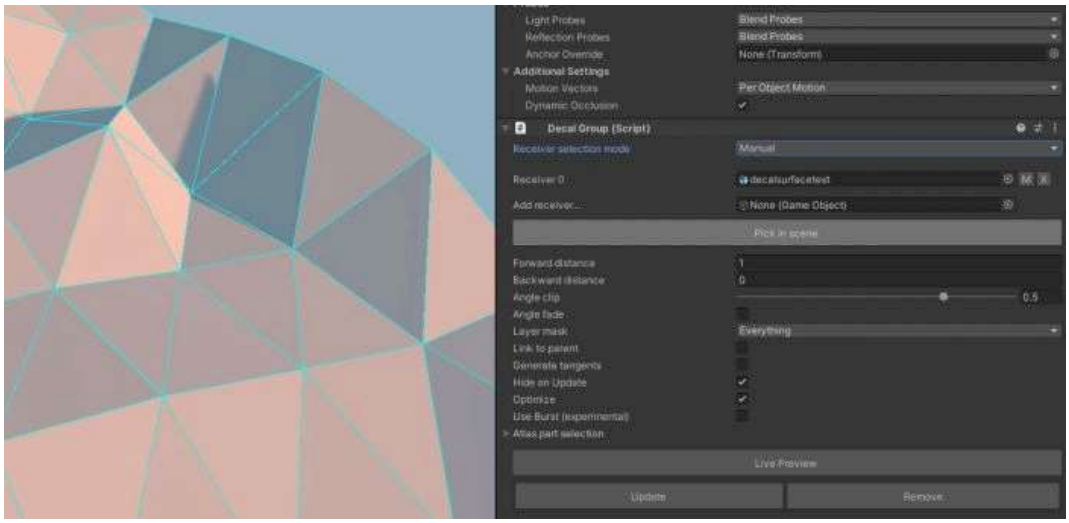


2. Set a decal-compatible shader on the decal mesh. Decalery includes some:

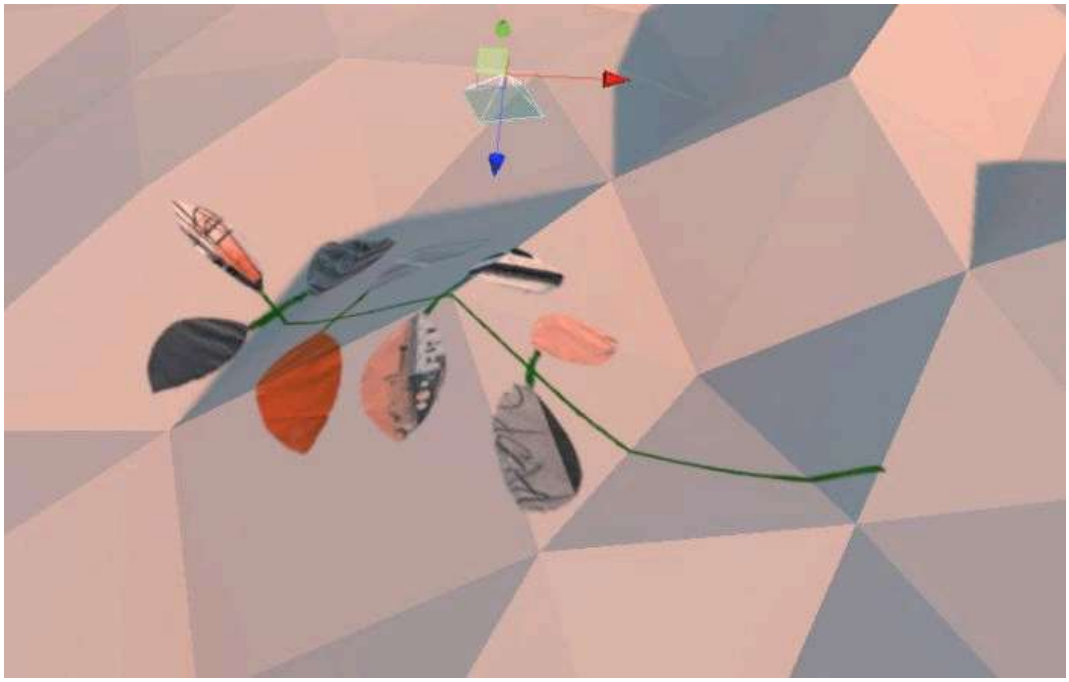
- DecalerySimple: just a basic textured decal with alpha channel; blend is configurable. Compatible with all render pipelines.
- DecaleryStandardPBR: Standard-like PBR shader for decals.
- DecaleryURPBasicPBR: Lit-like PBR shader for decals in URP.
- DecaleryHDRPBasicPBR: Lit-like PBR shader for decals in HDRP.

3. Add **Decal Group** component to it.

4. Click **Pick in scene**, then click on receiver meshes in Scene View. They will be added to the receiver list.



5. Click **Live Preview** and try moving the object around. Decal should become visible. If it's not, then it's probably not close enough to the receivers; increase **Forward distance** until the decal appears or move it closer.



6. Try placing more decals. You can also just click **Update** once instead of going to Live Preview, or click **Decalery -> Update all decals** to regenerate all decals in currently open scenes.

7. If lightmaps are rebaked, refresh decals by clicking **Decalery -> Update all decals**.

Decal order and layering

Since decals are simple meshes, their draw order is determined by their material Render Queue. Just decrease Render Queue on decal material to make it render below other decals. It is possible to have hundreds of decals layered over each other; the only limitation is the device fillrate (redrawing and reshading the same pixel over and over takes time on the GPU).

Terrains

Since v1.1, decals can be applied to terrains. Use the "Add receiver" field to manually add receiving terrains instead of the "Pick in scene" button. Final decal mesh generation performance depends on the original decal's size and projection distance. Small decals (such as bullet holes) will take less time than a huge decal covering a large portion of the terrain.

Decal geometry is built using the most detailed terrain LOD. To prevent lower terrain LODs from obstructing the decal when viewed from afar, increase Depth Offset on the decal material. If the offset is too high and the decal appears visibly closer than other objects standing on the terrain, use material Render Queue to ensure "Terrain -> Decal -> Other objects" order. Decals don't write to ZBuffer, so other objects will occlude it properly in this case.

Creating decals at runtime

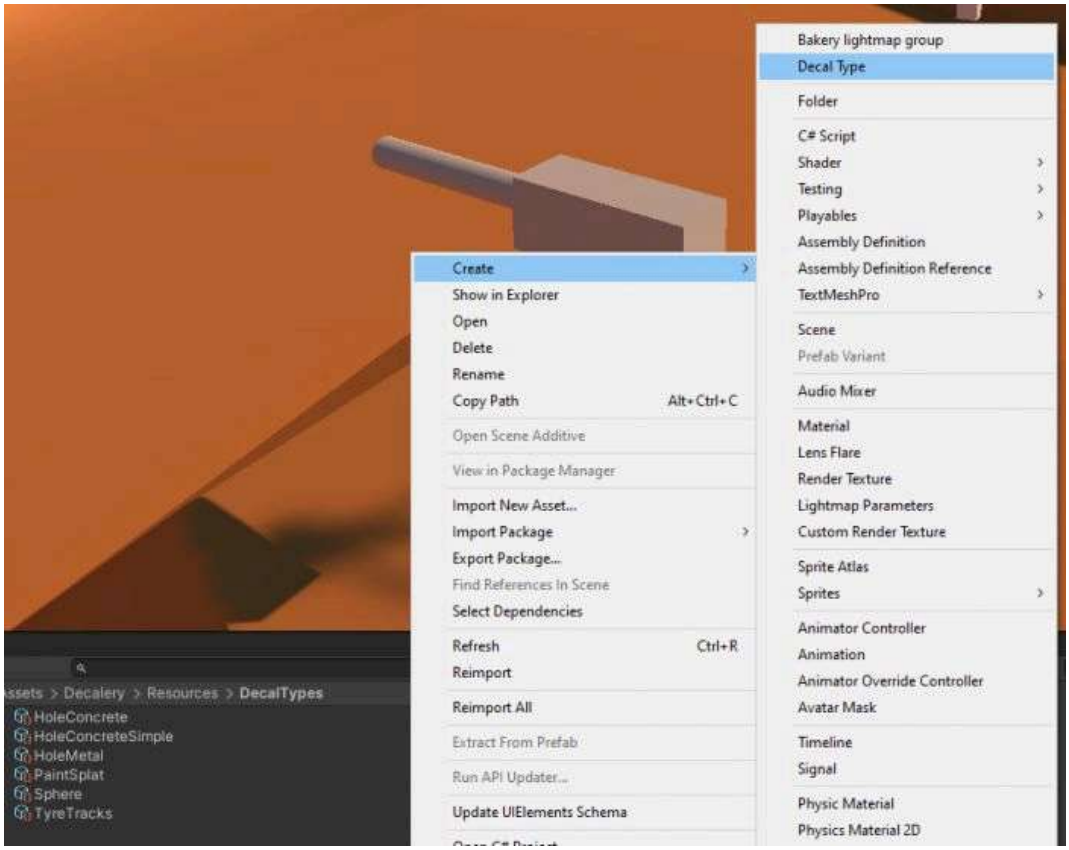
While it is possible to use the editor-style decals at runtime, Decalery also has a separate optimized system for in-game decals that is focused primarily on quad-like and trail-like geometry.

1. Add Decal Global Runtime Settings component to any game object. Assuming relatively modern (2019+) Unity, configure it like this for optimal quality/performance:

	Built-in render pipeline + Forward + DX11-12	Built-in render pipeline + Deferred + DX11-12	Built-in render pipeline + other GAPI	URP + DX11-12	URP + other GAPI	HDRP + DX11-12	HDRP + other GAPI
Mode	GPU With Readback	Full GPU	CPU Burst (install Burst and Mathematics packages)	GPU With Readback	CPU Burst (install Burst and Mathematics packages)	GPU With Readback	CPU Burst (install Burst and Mathematics packages)
Shader Pass	(not used, any value)	Deferred	(not used, any value)	(not used, any value)	(not used, any value)	(not used, any value)	(not used, any value)
Render Full GPU Mode With Built-In Render Pipeline	Off	On	Off	Off	Off	Off	Off

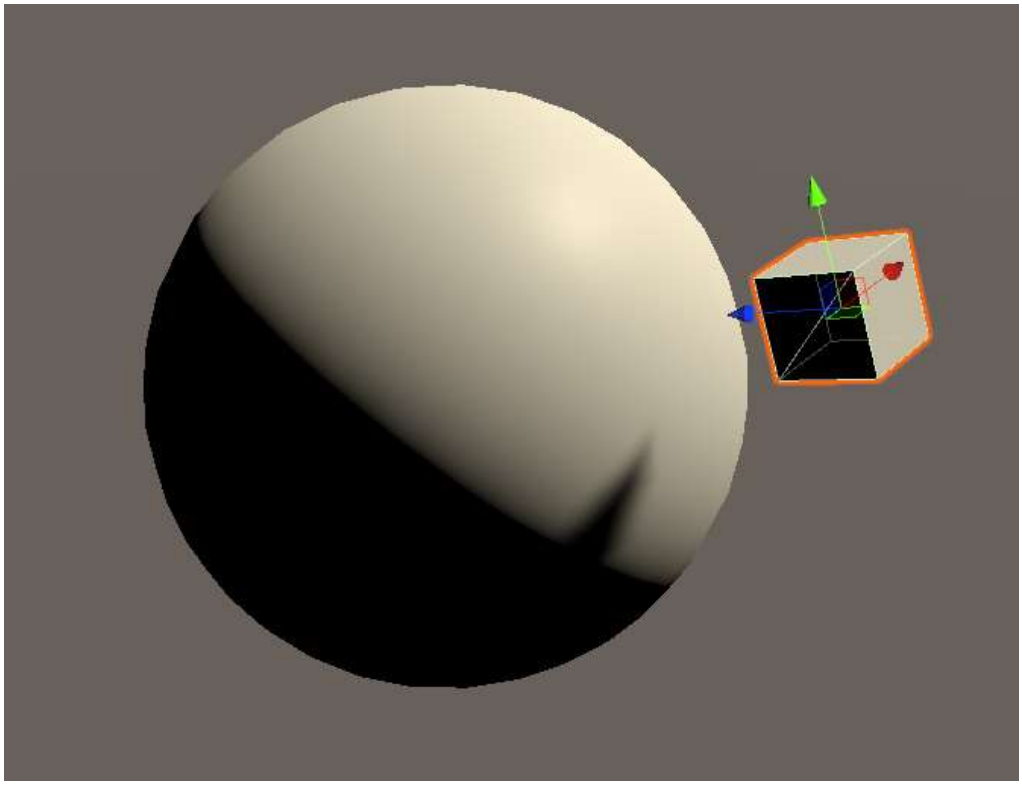
2. Create a material for your decal. It needs to use any of the decal-compatible shaders.

3. Create a Decal Type asset to describe your runtime decal:



4. Set decal material to Decal Type. Enable Tangents if decal needs to be normal-mapped.

5. Create a projector GameObject that will project decals. For this example, a simple cube will be used, with forward (blue) axis being used for projection:



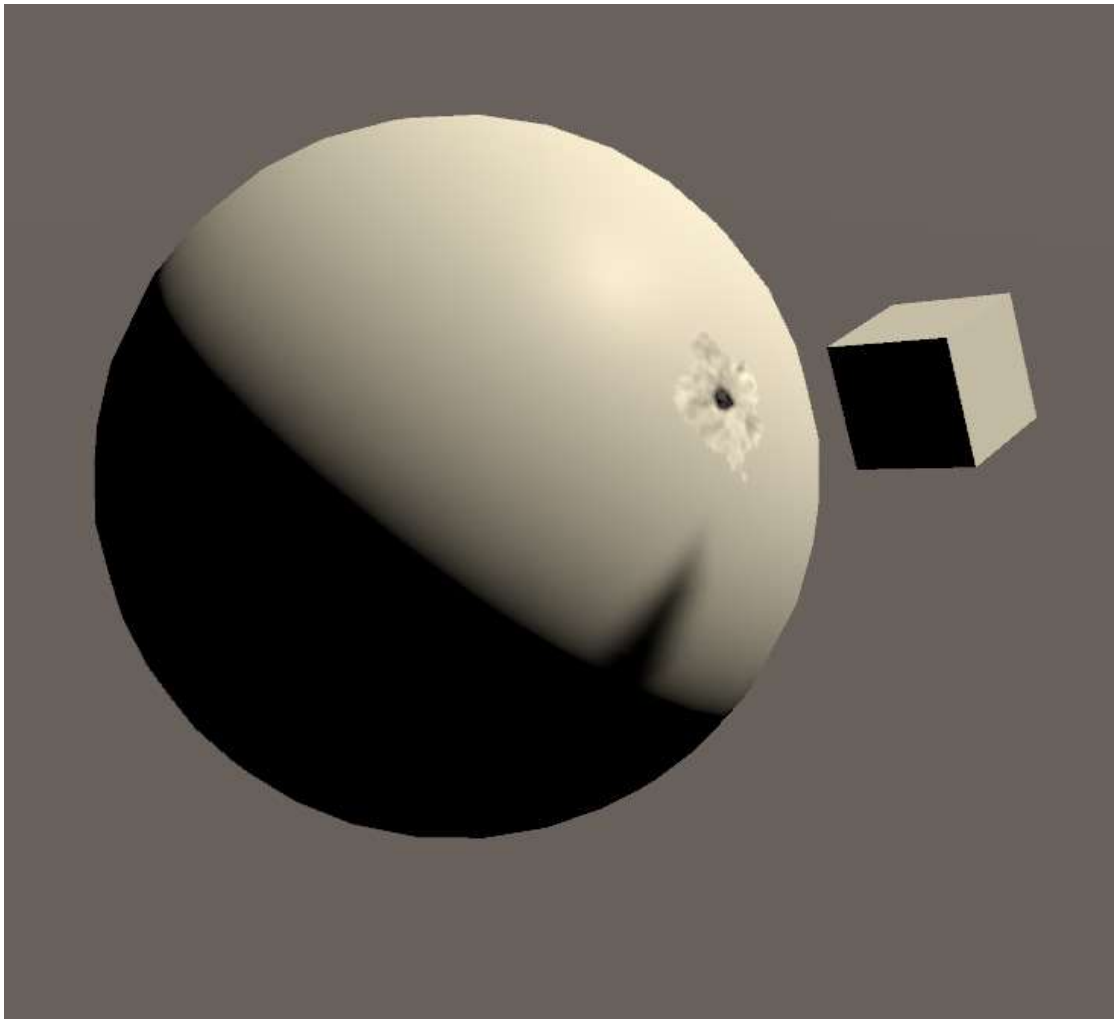
6. Add a script to this GameObject:

```
using UnityEngine;

public class DecalExample : MonoBehaviour
{
    public DecalType decalType;          // insert your Decal Type here
    public int maxTrisTotal = 4096;      // maximum triangles this spawner can generate for all decals using a particular lightmap ID or being parented to a
particular movable object
    public int maxTrisInDecal = 1024;    // maximum triangles per one decal
    public float decalSize = 0.2f;       // width and height of the decal
    public float forwardDistance = 1.0f; // forward projection distance
    public float opacity = 1.0f;         // opacity of the decal

    DecalSpawner spawner;

    void Start()
    {
        spawner = DecalManager.GetSpawner(decalType.decalSettings, maxTrisTotal, maxTrisInDecal); // Create Decal Spawner for our Decal Type
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space)) // wait for Space
        {
                RaycastHit hit;
                if (Physics.Raycast(transform.position, transform.forward, out hit)) // Raycast from our cube
                {
                        Transform rootObject = null;
                        if (hit.rigidbody != null) rootObject = hit.rigidbody.transform; // Consider object movable if it has a rigidbody and get its transform
                        spawner.AddDecal(transform.position, transform.rotation, hit.collider.gameObject, decalSize, decalSize, forwardDistance, opacity, 0,
rootObject); // Spawn the decal
                }
        }
    }
}
```

7. Insert your Decal Type into the script.

8. Make sure your receiver objects have colliders, so the script can raycast them.

9. Play the scene and hit Space. Decal should spawn: If nothing happens, and Mode is CPU, make sure that receiver models have Read/Write Enabled and Batching Static is turned off. For GPU modes this is not necessary.



For further examples see **example_decals_runtime** scene.

Current limitations

	Built-in (5.6)	Built-in (2017.X)	Built-in (2018.X)	Built-in (>= 2019.1)	Built-in (>= 2022.1)	URP	HDRP
Editor and Runtime CPU decals	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Editor and Runtime CPU decals (Burst-optimized)	No	No	No	Yes	Yes	Yes	Yes
Runtime decals: GPU with readback	No	Yes (slower mesh API)	Yes (slower mesh API)	Yes	Yes	Yes	Yes
Runtime decals: Full GPU	No	No	Yes, but local lights only work correctly in Deferred	Yes, but local lights only work correctly in Deferred	Yes, but only works correctly in Deferred	No (can't copy per-object data to decals)	No

DecaleryURPBasicPBR shader graph needs at least URP 7.

DecaleryURPExtendedPBR shader graph with additional Bakery features needs at least URP 12.

DecaleryHDRPBasicPBR and DecaleryHDRPExtendedGraph need at least HDRP 10.

Editor Decal Components

Decal Group

Represents a decal projector, defined by the MeshFilter/MeshRenderer on the same object. Decal shape can be anything from a simple quad, to a complex shape, such as e.g. a corner, a ring around another object, etc. MeshRenderer should be normally disabled after the decal is projected.

Decal gets position, normals and lightmap UVs from the receiver, but retains its own texture UVs, tangents and vertex color. Decalery assumes that vertex color is used as a single opacity value (fade).

Settings:

- Receiver selection mode: defines how the decal-receiving objects are selected. Possible options:
  - Manual: receivers are selected by hand. This is the most precise and predictable method. Objects can be chosen via both the standard Unity selector or by pressing the "Pick in Scene" button and directly clicking receivers in the scene. To the side of every added object there are **X** and **M** buttons:
    - Pressing X will remove the receiver from the list.
    - Pressing M will show a material field which will be used instead of the decal's own material only for the selected receiver. This is useful when, for example, the same decal is applied to a dry and a wet surface, where different shaders might be desirable.
  - Box Intersection: receivers are chosen by overlapping a box with nearby *colliders*.
    - Box scale: scales the selection box. Final box size is affected both by decal mesh bounds, *Forward/Backward distance* and *Box scale*.
  - Raycast from vertices: receivers are chosen by performing a physics raycast from every vertex of the decal's mesh, in the inverse normal direction. As with the previous option, it only picks up objects having colliders.
- Forward distance: decal projection distance (inside).
- Backward distance: decal projection distance (outside).
- Angle clip: maximum allowed angle between each decal triangle and projector triangle. The value is from -1 (-180) to 1 (180). This is to prevent texture stretching on surfaces parallel to the projection.
- Angle fade: fades vertex color/alpha to 0 as the projection approaches *Angle clip* value.
- Layer mask: a simple layer mask to filter out receiving objects.
- Parent mode: determines to which objects final decal meshes will be parented:
  - Scene Root: no parent;
  - Receivers: final decal meshes will be parented to their corresponding receivers (this option was previously known as *Link To Parent*);
  - Source: final decal meshes will be parented to the object holding this Decal Group.
- Hide on Update: disables source decal mesh renderer when Update or "Update all decals" are pressed.
- Generate tangents: does decal geometry need per-vertex tangents generated? Tangents are required for normal-mapping, parallax and other effects.
- Optimize: performs vertex welding and vertex order optimization.
- Create mesh asset: saves final decal mesh to an asset file instead of storing it inside the scene. Meshes will be saved to Assets/DecaleryMeshes/[sceneName]_[objectName].asset.
- Use Burst: uses Burst (https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html)-optimized code to generate decals. Requires Burst and Mathematics (https://docs.unity3d.com/Packages/com.unity.mathematics@1.3/manual/index.html) packages installed. Performs faster on high-poly geometry and in Live Preview.
- Atlas part selection: allows selecting a rectangular part of the decal texture to be used.

Buttons:

- Live Preview: when enabled, decal will be rebuilt interactively, as it is transformed, or its settings are tweaked.
- Update: (re)builds the decal geometry once.
- Remove: removes previously generated decal geometry.

Runtime Decal Components

Following components are only needed for in-game dynamic decals.

Decal Global Runtime Settings

Global settings for all runtime decals. Can be added on any object in the scene. Options:

- Mode: which apporach to use for runtime decal generation. Possible values:
  - **CPU**: same approach as for Editor decals (regular C# mesh generation code).
  - **CPU Burst**: faster Burst-optimized CPU generation code. Requires Burst (https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html) and Mathematics (https://docs.unity3d.com/Packages/com.unity.mathematics@1.3/manual/index.html) packages installed.
  - **GPU With Readback**: computes final decal mesh fully on the GPU, reads the result to RAM and stores into mesh.
  - **Full GPU**: computes final decal mesh fully on the GPU and draws it directly from computed buffers without any RAM readbacks.

There are different limitations to each technique.

First, there are **version-related limitations**.

On top of that, all **CPU** versions require meshes to be readable (Mesh.isReadable (https://docs.unity3d.com/ScriptReference/Mesh-isReadable.html)). This implies that receiver models must have **Read/Write Enabled** (https://docs.unity3d.com/Manual/FBXImporter-Model.html). Meshes drawn with static batching are not readable, and CPU decals cannot be applied to them at runtime.

GPU versions are currently only tested on DX11-12 (more supported GAPIs will come soon). They, however, do **not** require meshes to be readable and can work with batched meshes.

Full GPU mode uses DrawProceduralIndirect (https://docs.unity3d.com/ScriptReference/Rendering.CommandBuffer.DrawProceduralIndirect.html) to draw the decals. This way decals don't exist in the scene as regular GameObjects and require additional code to draw. Decalery has a built-in solution for such drawing in the built-in render pipeline.

Note that these limitations only apply to runtime (spawned in-game) decals - there are no platform limitations to Editor decals, i.e. Decal Groups.

	Editor decals	Runtime CPU / CPU Burst	Runtime GPU With Readback	Runtime Full GPU
Needs Read/Write Enabled	No	Yes	No	No
DX-only (currently)	No	No	Yes	Yes

- Shader pass: only relevant in Full GPU mode. Chooses decal shader pass to use. Possible values:
  - Forward base: use in forward rendering.
  - Deferred: use in deferred rendering.
- Render Full GPU Mode With Built-In Render Pipeline: this option adds command buffers to currently active camera to draw decals in Full GPU mode (only applicable to Standard render pipeline and if Full GPU mode is chosen).

Decal Mesh Ref

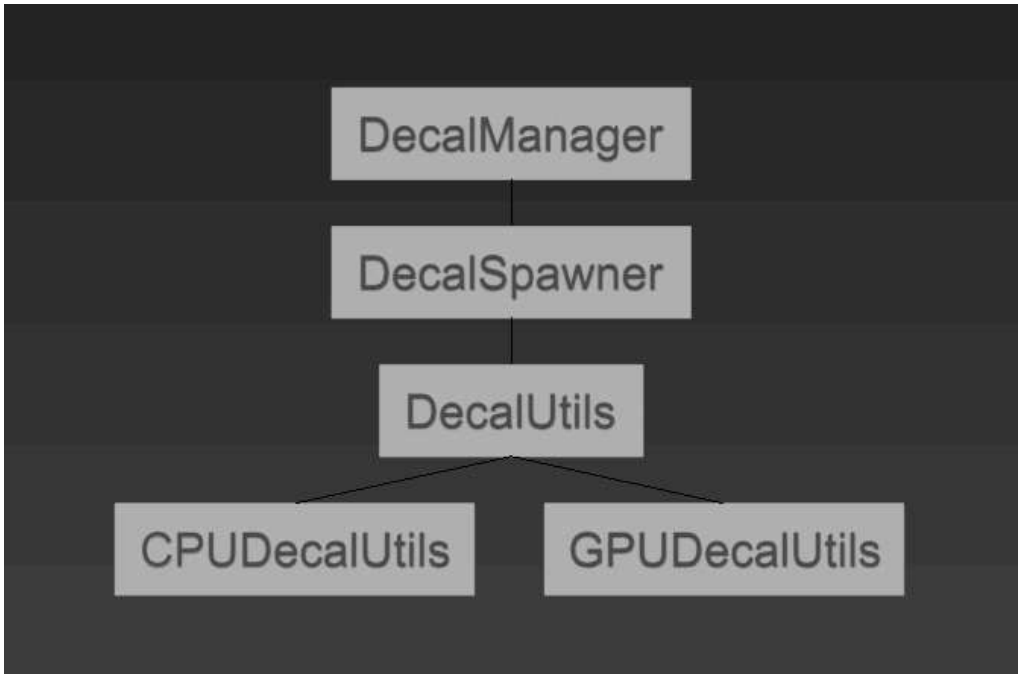
When using raycasts to spawn decals, sometimes there is a problem of colliders and renderers being on completely different objects. A typical example is a skinned character which may have colliders on bones, but the skinned mesh itself is separate. Decal Mesh Ref component can be added on collider objects to link them to corresponding **Renderers**. Decal Spawner will automatically detect these objects and redirect decals.

Menu

- **Decalery -> Update all decals**: regenerates all decals in open scenes.
- **Decalery -> Optimize batch count**: combines decal meshes to reduce draw call (batch) count. Final decal meshes are further combined if they:
  - Share the same material;
  - Share the same parent;
  - Share the same lightmap;
  - Are not further apart than the "Limit world batch size" value;
  - Do not combine to more vertices than the "Limit vertices per batch" value.
- **Decalery -> Select decal source**: if the final decal mesh is selected, will change selection to its source object (the one with the Decal Group).

Runtime API

Decals can be created at runtime. While it is possible to directly use the DecalGroup component, it is not recommended for performance reasons. Decalery has a special fast mode of generating and rendering simple quad-like and trail-like decals at runtime. The architecture is as follows:



DecalSpawner

DecalSpawner class is the main building block that manages pools of decals using identical settings and materials. For example, in a shooter, a single DecalSpawner for bullet holes can be used by all guns of all characters. Internally DecalSpawner will create lower-level *DecalUtils.Group* objects for every combination of shader variant / lightmap index / parent transform.

Properties:

- **`InitData initData`**
  Main spawner settings. InitData class contains following members:
  - **`Material material`**
    Base material used by all spawned decals. Use shaders made specifically for decals (offsetting position to camera to prevent Z-fighting).
  - **`bool isTrail`**
    Is this decal a trail (e.g. a tire track)? Trails connect edge-to-edge instead of being separated quads. Trails also have a unique continuous UV generation style.
  - **`float trailInterval`**
    If *isTrail* is enabled, controls interval between trail edges (smaller interval = rounder trails).
  - **`float trailVScale`**
    If *isTrail* is enabled, controls vertical texture coordinate tiling.
  - **`bool tangents`**
    Should decals have tangents (do they need normal mapping)?
  - **`bool inheritMaterialPropertyBlocks`**
    Should these decals inherit MaterialPropertyBlocks from receivers?
  - **`bool useShaderReplacement`**
    Use shader replacement feature? (see next)
  - **`ShaderReplacement[] shaderReplacement`**
    Allows optionally overriding decal shader based on the receiver's shader. Can be used when applying decals to e.g. special surfaces with vertex deformation. ShaderReplacement structure contains following members
    - **`Shader src`**
    - **`Shader dest`**
      If receiver has shader *src*, decal will use shader *dest*.
- **`Dictionary<int, DecalUtils.Group> staticGroups`**
  A map of all static (that is, not having any parent transform) *DecalUtils.Group*s created by this spawner, indexed by their lightmap ID.
- **`Dictionary<Transform, DecalUtils.Group> movableGroups`**
  A map of all movable *DecalUtils.Group*s created by this spawner, indexed by their parent transform.

Methods:

- **`void Init(int maxTrisTotal, int maxTrisInDecal, DecalUtils.Mode preferredMode, bool preferDrawIndirect, int preferredShaderPass)`**
  Initializes the spawner using its properties. Must be called only once.
  - **`int maxTrisTotal`**
    Maximum amount of triangles used in a single *DecalUtils.Group* spawned. Decal triangle count depends on receiver geometry detail and decal size. Older decals will disappear when new decals are added above the limit.
  - **`int maxTrisInDecal`**
    Maximum allowed triangle count for one decal. Similarly, it depends on receiver geometry detail and decal size. While this number can be set to *maxTrisTotal* for safety, using a lower realistic value will reduce processing time.
  - **`DecalUtils.Mode preferredMode`**
    Preferred decal generation mode. Possible values are:
    - **`DecalUtils.Mode.CPU`**
      Generation is done on the CPU, similar to how the DecalGroup component works. This is the slowest, but also the most platform-compatible option. Requires receiver meshes to have Read/Write Enabled (https://docs.unity3d.com/Manual/FBXImporter-Model.html).
    - **`DecalUtils.Mode.GPU`**
      Generation is done on the GPU. This option requires hardware support for Unordered Access Views and Geometry Shaders (i.e. sm5_0). Receivers do **not** need to have *Read/Write Enabled*.
  - **`bool preferDrawIndirect`**
    Are these decals supposed to be rendered with DrawProceduralIndirect (https://docs.unity3d.com/ScriptReference/Rendering.CommandBuffer.DrawProceduralIndirect.html)? Such decals do not require costly VRAM->RAM->VRAM memory transfers, as the generated buffer is used directly by the drawing shader. Shader must be aware of this method.
  - **`int preferredShaderPass`**
    Shader pass used by indirect-drawing shaders. If the pass is not present in the shader, it will be clamped to the highest available value.

> Currently, skinned meshes only support CPU generation mode. DecalSpawner will automatically fallback to CPU when used on skinned meshes.

- **`void AddDecal(Vector3 position, Quaternion rotation, GameObject hitObject, float decalSizeX, float decalSizeY, float distance, float opacity, float angleClip, Transform rootObject)`**
  Spawns a decal on *hitObject* from a projector placed at *position* in the direction of *rotation*.
  - **`Vector3 position`**
    Projection origin.
  - **`Quaternion rotation`**
    Projection rotation. Decals are projected along the positive Z axis.
  - **`GameObject hitObject`**
    Object receiving the decal. Must have either a *MeshRenderer*, a *SkinnedMeshRenderer* or a *DecalMeshRef* (containing a list of renderers). When the decal is applied to a *SkinnedMeshRenderer*, *hitObject* (bone) matrix will be used to transform into bind pose correctly.
  - **`float decalSizeX`**
    Decal width.
  - **`float decalSizeY`**
    Decal height.
  - **`float distance`**
    Decal projection distance (same as Forward distance in DecalGroup).
  - **`float opacity`**
    Opacity multiplier.
  - **`float angleClip`**
    Same as *Angle clip* in DecalGroup.
  - **`Transform rootObject`**
    Optional transform used as a parent for this decal. Assign movable objects' transforms here.

- **`void AddDecalToQueue(Vector3 position, Quaternion rotation, GameObject hitObject, float decalSizeX, float decalSizeY, float distance, float opacity, float angleClip, Transform rootObject)`**
  Similar to *AddDecal*, but adds the decal to the queue instead (see next).

- **`void UpdateQueue(int maxDecalsPerFrame)`**
  Takes *maxDecalsPerFrame* elements from the queue and executes them. This is useful to prevent too many decals spawning in one frame, thus affecting performance.

- **`void Clear()`**
  Removes all visible decals, created with this spawner. Doesn't release memory.

- **`void Release()`**
  Removes all visible decals, created with this spawner and releases all used memory.

Static properties:

- **`static List<DecalSpawner> All`**

A list of all spawners ever created.

DecalManager

DecalManager manages the creation and reuse of DecalSpawners. Its main purpose is to let different scripts to access the same shared spawners.

Static methods:

- **`static void SetPreferredMode(DecalUtils.Mode mode, bool drawIndirect, int pass)`**
  Sets preferred generation/rendering settings for all newly created spawners (see *Init*).
- **`static DecalSpawner GetSpawner(DecalSpawner.InitData initData, int maxTrisTotal, int maxTrisInDecal)`**
  Gets an existing spawner or creates a new one using provided settings.
- **`static DecalSpawner GetSpawner(string name, int maxTrisTotal, int maxTrisInDecal)`**
  Same, but will attempt to load *initData* from *Resources/DecalTypes/name.asset*. This data can be stored in *DecalType* assets, as seen in the included example scenes.
- **`static DecalSpawner CreateUniqueSpawner(DecalSpawner.InitData initData, int maxTrisTotal, int maxTrisInDecal)`**
  Similar to *GetSpawner*, but always creates a new spawner. Normally. this is only useful for trail decals, as they connect to each other within one *DecalUtils.Group*.

DecalUtils

DecalUtils.Group is a lower-level object, representing a fixed-sized pool of decals, sharing a common material, lightmap index and parent transform. Most of the time there is no need to use it directly. *DecalUtils* class uses static methods to work with *DecalUtils.Group*. Underneath, most methods branch to either *CPUDecalUtils*, *CPUBurstDecalUtils* or *GPUDecalUtils*.

CPUDecalUtils

CPUDecalUtils are used internally for both runtime CPU decals and in-editor DecalGroups.

CPUBurstDecalUtils

CPUBurstDecalUtils is the Burst-optimized version of CPUDecalUtils.

GPUDecalUtils

GPUDecalUtils are used internally for runtime GPU decals. There are some GPU-specific methods that are useful for indirect drawing:

- **`static CommandBuffer CreateDrawIndirectCommandBuffer(DecalUtils.Group group, int pass)`**
  Creates a CommandBuffer (https://docs.unity3d.com/ScriptReference/Rendering.CommandBuffer.html) drawing the decals directly from generated data on the GPU, with the specified shader pass. Decal group must be created witn *indirectDraw* enabled.
- **`static void UpdateDrawIndirectCommandBuffer(DecalUtils.Group group, Matrix4x4 mtx, int pass)`**
  Updates the command buffer using a new transformation matrix.

Examples of GPUDecalUtils usage can be found in *DecalGlobalRuntimeSettings.cs*.
