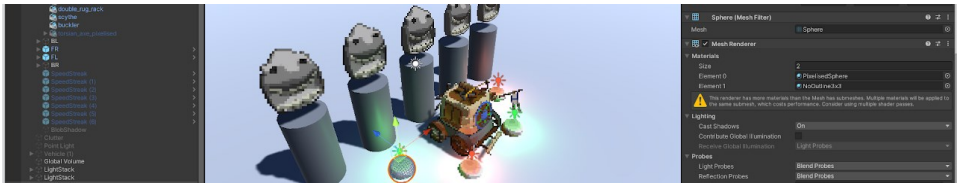
User Guide (v1.5)



Requirements

The package has the following minimum requirements*:

- Universal Render Pipeline 7.3.1

Please let me know if you find it working without these requirements, or not working with them!

I have tested using the following versions of Unity:

- ProPixelizer v1.5
  - 2019.4.28 LTS + URP 7.6.0
  - 2020.3.13 LTS + URP 10.5.0
- ProPixelizer v1.4
  - 2019.4.23f1 (LTS) + URP 7.3.1
  - 2020.2.1f1 + URP 10.2.22
- ProPixelizer v1.3
  - 2019.3.0 + URP 7.4.x
  - 2020.1.2f1 + URP 8.2.0
  - 2020.2.1f1 + URP 10.2.22

Build Targets

I've tested:

- webGL ES 2.0 (firefox and chrome), running on a desktop.
- Windows PC, both Direct X and openGL.
- Android (works, but laggy on my low end phone, a Galaxy A10).

If you are interested in how ProPixelizer works, I give a brief description of the pixelization process in this article, under Attempt #3. I intend to write more technical articles in the future. Feel free to contact me on Discord or twitter with questions!

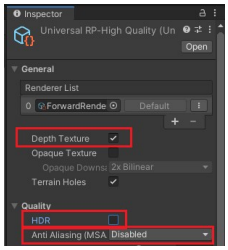
Getting started

A really quick video tutorial can be found here: https://www.youtube.com/watch?v=S9tT6Fu1lRQ

Below is a more detailed step-by-step guide on how to set up a project to use ProPixelizer.

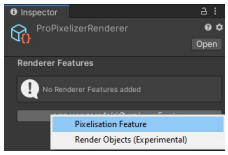
First, make sure you have Universal Render Pipeline package added to the project, then add the ProPixelizer package.

Setting up the render pipeline



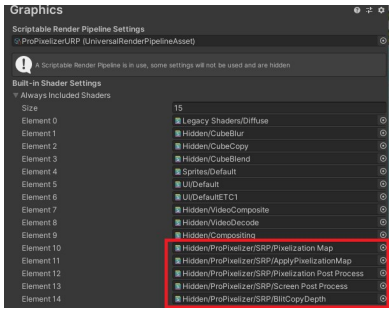
In the Universal Render Pipeline Asset, enable *Depth Texture* and disable *HDR*.

Disable *Anti Aliasing*.



Select the Renderer asset (here - ForwardRenderer, in the RendererList). Below *Render Features* click *Add Render Feature*, and select *Pixelise Feature* to add the render pass required for ProPixelizer.

Setting up the project

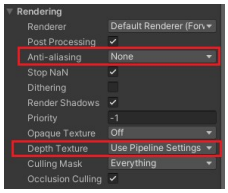


Under `Project Settings > Graphics`, add the following ProPixelizer shaders to the Project's *Always Included Shaders*:

- Hidden/ProPixelizer/SRP/Screen Post Process
- Hidden/ProPixelizer/SRP/Pixelization Post Process
- Hidden/ProPixelizer/SRP/BlitCopyDepth
- **v1.3+:** Hidden/ProPixelizer/SRP/Pixelization Map
- **v1.3+:** Hidden/ProPixelizer/SRP/ApplyPixelizationMap

This step is required because some versions of Unity have an issue detecting that these shaders are used by the Project - because they are only referenced in the code (specifically, in the Pixelise Feature*) - and so Unity mistakenly omits them from the build. It's only required for building your project, and you can skip it for the moment if you are in a rush to get started.*

Setting up the camera



Configure the camera as follows:

- *Rendering/Anti-aliasing*: None
- *Rendering/Depth Texture*: Use Pipeline Settings
- *Output/HDR*: Off

Post processing can be used, eg *Bloom* and *Vignette*, and configured through a Volume as per the usual Unity workflow. Note that *Depth of Field* is currently unsupported (but will be added in a future release).

For best results, it is strongly recommended to **use orthographic rather than perspective** projection. This is because pixel creep can only be fully eradicated for cameras using orthographic projection; for orthographic projections, the size of an object does not change as it moves within the camera's view.

Add the **Camera Snap SRP** MonoBehaviour to the camera game object.

Note that the camera's world-space pixel size will be set by the **Camera Snap SRP** component during runtime.

Updating from previous versions

Updating to **v1.4**:

- I changed the name of some shader keywords to be consistent between outline and appearance materials. After you update, run the tool *Window/ProPixelizer/Verify Materials* to fix any broken materials in your project (this will update all materials to use the new keywords).

Pixelized objects: materials setup

Objects can be pixelized into *macropixels* of size 1x, 2x, 3x, 4x, and 5x screen pixels.

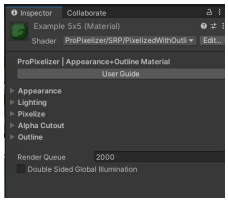
ProPixelizer draws pixelated objects by applying two passes:

- An **appearance pass**, which reduces the color, adds dither if required, and quantises the shading.
- An **outline pass**, which fills a buffer with information required to draw object outlines (outline color, object ID).

You have two options to add these passes:

1. The *recommended way* is to use the ProPixelizer **PixelizedWithOutline** material shader, which adds all required passes to the object.
2. You can also add the passes by adding separate **Appearance** and **Outline** materials.

Example assets are included with ProPixelizer to show how materials can be configured.



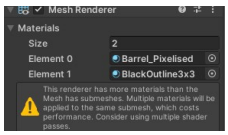
PixelizedWithOutline material

The PixelizedWithOutline material adds all passes required to draw and outline the object. It uses a custom editor inspector and categorizes the shader properties as follows:

- **Appearance:** Properties for albedo maps, normals maps, emission maps, and whether to color grade the object using a palette.
  - The **Palette** texture property is a texture look-up table (LUT) used for color grading when the 'Use Color Grading' option is enabled. A number of LUTs to emulate well-known devices (eg NES, PAL, GameBoy) are shipped with **ProPixelizer**.
  - You can create your own palettes by using a palette asset (Right Click -> Create -> ProPixelizer -> Palette) which is used to generate the LUTs. The LUT can also include dither patterns as demonstrated here. For more information, see the 'Color palettes and dither patterns' section of the user guide below.
- **Lighting:** Shadow and lighting ramp.
  - **Lighting Ramp** is a texture ramp used for cell shading, which helps give a pixel-art aesthetic. When rendering the object, the calculated color Value is used as to sample the lighting ramp.
  - **Ambient Light:** A background level of lighting. This can be helpful to tweak the appearance, in conjunction

with the color grading.

- **Pixelize:** Properties to pixelate the object.
  - **Pixel size** in the range 1-5. This sets the size of one 'macropixel' in screen pixels.
- **Alpha Cutout**: Properties to enable cutout transparency.
- **Outline:** Outline related properties.
  - **ID:** A number. Outlines are drawn when pixels have an ID different to those around them. If two objects should have outlines when they meet, give them different IDs (eg, two enemies). If they should not have outlines (eg, a character and their equipment), give them the same ID. The value should be an integer in the range (0, 255).
  - **Outline Color:** The color to use for the outline. The alpha value controls blending with the scene color. Alpha values of 0 can be used for invisible outlines, 1.0 can be used for block color, and fractions to blend with the appearance material color.



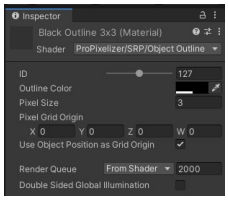
Separate appearance and outline materials

The image on the left shows an example setup to correctly render an object as pixelated when using separate appearance and outline materials. The warning can be ignored - the outline material only defines a shadowcasting pass and a pass used for rendering metadata. It does not lead to performance loss during normal rendering passes.

Even if no outline is desired, you should still add an outline material - just set the outline color to an alpha of 0.

Appearance material

Properties are identifcal to the PixelizedWithOutline material, but without the outline properties.



Outline material

The outline material has the following properties:

- **ID:** A number. Outlines are drawn when pixels have an ID different to those around them. If two objects should have outlines when they meet, give them different IDs (eg, two enemies). If they should not have outlines (eg, a character and their equipment), give them the same ID. The value should be an integer in the range (0, 255).
- **Outline Color:** The color to use for the outline. The alpha value controls blending with the scene color. Alpha values of 0 can be used for invisible outlines, 1.0 can be used for block color, and fractions to blend with the appearance material color.
- **Pixel Size/Pixel Grid Origin:** These are the same as in the appearance material. **For almost all situations, you should make sure these values match those in the appearance material.**

An outline material must be added to the object even if the outline is set to invisible; the outline material is used to render metadata required for outlines and the pixelisation of objects.

Technical note for advanced users: If you are writing your own ShaderLab shaders, you can perform the pixelisation and outline within a single material shader. In the fragment shader, you should `clip()` the output of `PixelClipAlpha_float` in `PixelUtils.hlsl`, and also add a separate Pass with Tags `"LightMode" = "Outlines"`, which you can copy directly from the `ObjectOutline.shader`. The **only** reason a separate material is used by default for the outlines in ProPixelizer is to maintain support for ShaderGraph materials, which currently do not allow me to add a separate pass to the shader. In ProPixelizer v1.4 I added a multi-pass shader that uses passes from the ShaderGraph shader and the outline shader.

Eradicating pixel creep

Pixel creep is a problem that occurs frequently when rendering 3D objects as pixel art - the object appears to shimmer as it moves across the screen. An example can be seen in this video: https://www.youtube.com/watch?v=lO8O5tY-wZI . The creep occurs because the number of pixels that an object occludes changes as it moves across the screen.

Pixel creep can be removed by aligning objects to the pixels of the screen before rendering them. **Note that pixel creep can only ever be solved for orthographic projections**; in a perspective projection, the object's size will change as it moves on the screen.

For orthographic projections, ProPixelizer provides functionality to handle this for you, through two MonoBehaviours:

- A `CameraSnapSRP` MonoBehaviour, which should be attached to your camera. The `PixelSize` property determines the size of one camera pixel in world units.
- An `ObjectRenderSnapable` MonoBehaviour, which should be attached to meshes that you are rendering.

These MonoBehaviours will snap object positions before rendering and restore them afterwards. The implementation respects transform hierarchies.

Snapping angles

The `ObjectRenderSnapable` MonoBehaviour can also snap the angles of objects being rendered. Set `ShouldSnapAngles` to true if object angles should also be snapped, by the desired `Angle Resolution`. If you have nested transformations (eg for equipment), I recommend you only do this on the root transform.

Aligning pixel grids

The `ObjectRenderSnapable` MonoBehaviour also provides a way to align the pixel grids of child objects to their root transform. This is useful for things like equipment - you have different meshes but want the complete object to look like one sprite. Set the property `Use Root Pixel Grid` to true to cause the object's pixel grid to be aligned with the root transform of the hierarchy.

An example is given in the picture on the left, showing how the blue/green shield looks when it is and is not aligned to the pixels of the root transform. Look closely within the circles - on the unaligned you should be able to see the slightly 1 pixel misalignment of the two objects with 3x3 pixelsizes.

More control over object outlines

Outlines are drawn whenever two adjacent pixels have a different `ID`, as determined from the `Object Outline` material property.

The `OutlineControl` MonoBehaviour provides some extra ways to control the outlines of your objects:

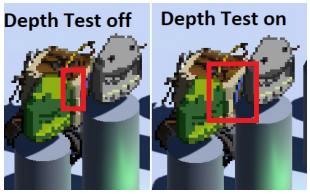
- You can specify an `ID` to use, or set `Use Random ID` to true to generate a random ID at runtime.
- Set `UseRootID` to true if the outline material should use the ID of the root transform. This also requires an `OutlineControl` MonoBehaviour to be present on the root transform. Like with aligning the pixel grids, this is useful for child objects such as equipment attached to a player model - it ensures the outline is drawn around the entire player, and not around each individual piece of equipment. In the image on the left, I have

set `UseRootID` to true on all attached equipment (eg the shield, crossbow, window, etc) but not around the wheels.

- The same is also true of color. You can specify the color to use for the outline with `Color`, and also whether to `UseRootColor`.

Note that many of these changes will only take place once you hit play mode (they require instancing the material).

Technical note for advanced users: The `ID` is rendered into the outline buffer with 8-bit precision, so only 256 different values of ID can be specified.



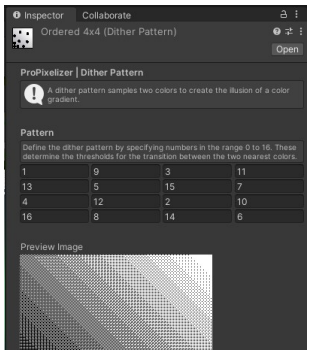
Depth Test outlines

This option was added in **v1.3** and can be found on the `PixelisationFeature` that you added to your `ForwardRenderer` Asset in the *Getting Started* section. When enabled, outlines will only be drawn for edge pixels that are in front of their neighbors. This helps achieve the feel of a hand-drawn sprite, for which the outline would not change depending on the geometry in front of it. A comparison of the two settings is shown in the image here.

Color palettes and dither pattern tools (beta!)

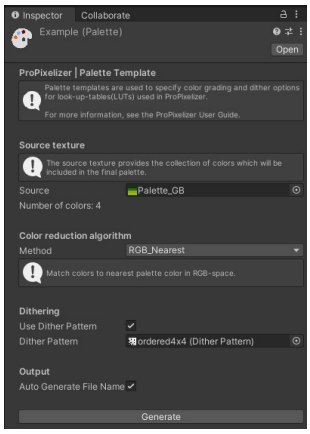
Many old games used reduced color palettes, often due to hardware limitations! For example, the original GameBoy could only display 4 different brightness values, and the SNES could only display 256 colors at once. Games sometimes employed dithering to emulate an increased color depth.

ProPixelizer gives you a set of tools to create reduced color palettes and dither patterns. Both are combined into a texture Look-Up Table (LUT) which is sampled when rendering the object to color grade with minimal overhead.



Dither patterns

ProPixelizer supports 4x4 dither patterns. You can design your own dither pattern by creating a dither pattern asset (Create -> ProPixelizer -> Dither Pattern). The editor will display a 4x4 grid of values, and a preview of how the dither pattern will appear for a smooth monochrome gradient. Each 'value' in the grid ranges from 0 to 16, and shows the 'threshold' at which this pixel in the 4x4 pattern will be enabled. Some example patterns are included (ProPixelizer/Palettes/DitherPatterns).



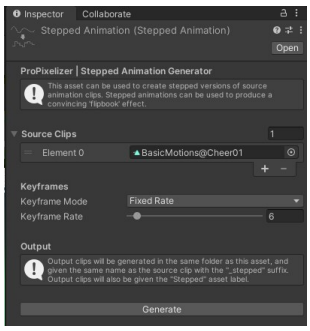
Color palettes

You can configure your own palettes using a palette asset (Create -> ProPixelizer -> Palette). After configuring the palette, use the 'Generate' button to create a texture LUT that you can use in your ProPixelizer materials.

A number of properties can be configured:

- **Source Texture:** The texture to sample the set of colors from. The text below will tell you how many colors were identified in the source - these are used for color matching when generating the look up table.
- **Color reduction algorithm:** The method to use when deciding which colors are most similar. The comparison can be made in RGB-space, HSV-space, or just using the 'value' of the HSV space (which can work well for monochrome palettes in which hue does not matter).
- **Dithering:** Whether to use a dither pattern.
- **Output:** Options for generated LUT file name.

Stepped animation tools (beta!)



Traditional pixel-art games used hand-drawn sprite sheets, and characters were animated by changing which sprite was displayed - the result was like a 'flipbook', with a few well defined poses. For instance, the cycle for a run animation could be ~5 frames long. 3D animations, on the other hand, commonly interpolate between key frames to produce smooth movement.

To help users achieve a flipbook effect, ProPixelizer provides a utility for automatically converting animation clips into stepped versions. A copy of the animation clip is created, keyframes are decimated, and the interpolation is set to None. This utility allows you to use many standard animations (e.g. from the Asset Store) in a way that keeps the pixel art aesthetic.

To use these tools, create a Stepped Animation asset (Create -> ProPixelizer -> Stepped Animation). After configuring, click 'generate' to create copies of the source clips.

Lastly, if you are using clips with anim trees, you may also wish to turn off the blending for the anim tree - otherwise changes of state will interpolate between frames, and break the flipbook feel.

Other tips for achieving a good pixel-art feel

Below are a collection of tips to bear in mind when using ProPixelizer:

- Try to avoid small features in geometry that are less than a pixel in size - otherwise they can flicker in and out of visibility.
- Old school sprite art typically only has a few different viewing angles - snap the angles of your objects to achieve the same feel.
- Reduce the number of keyframes in animation and use 'constant' interpolation to give it a stepped feel, as if flicking through pages of a sprite sheet.
- When using Color Grading, it helps to create your assets while targeting a particular color palette. There are significant differences between the various old-school color palettes. GameBoy is monochrome, for instance, while PAL is dark and grungy.

Note that you can also do neat things not normally seen in pixel art games, such as

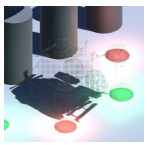
depth-of-field post processing.

I hope you enjoy using ProPixelizer. Please do message me if you have any questions or problems. I'd also be very pleased to hear what you make with it.

Cheers!

Elliot

FAQ



My object looks like a bunch of dots?

The problem: Pixelated objects appear as a bunch of dots when rendered, as if dithered.

The solution: The Pixelisation Feature needs to be added to the Render Features of the Scriptable Render Pipeline - see 'Setting Up The Render Pipeline' under 'Getting Started', above.

If you are interested in *why* it looks like this, you can find the answer in this Medium article which describes how ProPixelizer works. The method used for ProPixelizer is described under Attempt #3; objects are first drawn as a dithered matrix of dots, then the post process fills the surrounding screen pixels to produce the final pixelated image.

If you see a shader error in the console when compiling the ScreenPostProcessing shader, please email me with details! The dots will also appear when the post process is not working, which is a bug.

Help, I see a black screen in builds/nothing shows in my build!

Make sure that Unity is correctly **adding the ProPixelizer post-process shaders to your build** - in some configurations, Unity mistakenly assumes these shaders are not required and strips them. See 'Setting up the Project' in this guide for the configuration of 'Always Included Shaders' to use.

If you have multiple quality configurations (eg, different assets for low and high quality targets), make sure each Universal Render Pipeline Asset is also set up correctly as in the first section of this guide.

I have a mesh with multiple materials - how should I apply the appearance+outline materials?

- From **ProPixelizer v1.4+**, you can now apply a combined Pixelized+Outline material using the *ProPixelizer/SRP/PixelizedWithOutline* shader.

Update history

Version 1.5

- **Added** improved workflow for palettes. Individual palettes are now assets, so the values used for generating LUTs are saved.
- **Added** editor tool for user-defined 4x4 dither patterns.
- **Added** editor tool for creating stepped animations, to help replicate a flipbook/spritesheet feel.
- **Fixed** Bug for game view on Metal.
- **Fixed** Bug when RenderScale != 1.
- **Fixed** Correct pixelation for material preview, material icons and shader graph preview (fix for 2020+ only, URP on 2019 LTS doesn't expose required properties).
- **Fixed** editor warning when selecting in inspector (fix for 2020+ only, URP on 2019 LTS doesn't expose required properties).

Version 1.4

- **Added** dither pattern support.
- **Added** single material to produce both outlines and appearance.
- **Added** example scene (*Floating*) to show new shader and a no-creep setup.
- **Fixed** occasional 'tearing' due to numerical precision error.
- **Fixed** creep in rare cases.
- **Improved** Palette Builder tool.
- Made shader keywords consistent - run *Window/ProPixelizer/Verify Materials* to fix broken materials.

Version 1.3

- **Added** option for depth-tested outlines, to prevent outlines when objects overlap (see option in render feature).
- **Added** alpha cutout support.
- **Added** custom GUI for Appearance materials (for ShaderGraph versions that support it, 2020+).
- **Fixed** Object flashing bug for Orthographic projections when near plane was negative.
- **Fixed** gaps at screen edge.
- **Fixed** various truncation warnings.
- **Fixed** support for Unity 2020.2 and URP 10.2
- **Performance improvement:** Large performance improvement on all platforms (3 pixelation passes for color/outline/depth now reduced to a single 'Pixelization Map' pass).

Version 1.2

- **Fixed** creep/malformation at some resolutions.
- **Fixed** tearing in perspective projection.
- **Fixed** 3x3 pixelation on AMD+openGL targets.
- **Quality of life:** the '_ID' property in the ObjectOutline shader is now specified as integer in the range (0,255). No change is required if using the OutlineControl MonoBehaviour.

Feel free to email with feature requests and suggestions!