Jordan Roth

Photo Editor V2, 2018

Photo Editor

Table of Contents

[Quick Intro 3](#_Toc508968458)

[How to Contribute (Needs Finished - links) 3](#_Toc508968459)

[Asset Store (click for link) 3](#_Toc508968460)

[GitHub (click for link) 3](#_Toc508968461)

[Itch (click for link) 3](#_Toc508968462)

[Quick Start 4](#_Toc508968463)

[Step 1 - Imports 4](#_Toc508968464)

[Step 2 – Global Variables 4](#_Toc508968465)

[Step 3 – Start Method 4](#_Toc508968466)

[Step 4 – Check File 4](#_Toc508968467)

[Step 5 – Find Image 5](#_Toc508968468)

[Step 6 – Image Settings 5](#_Toc508968469)

[Step 7 – Add Plane 6](#_Toc508968470)

[Step 8 – Add Script 6](#_Toc508968471)

[Step 9 – Set Lighting 6](#_Toc508968472)

[Step 10 – Hit Play 7](#_Toc508968473)

[Understanding the Scripts 8](#_Toc508968474)

[Texture2DEditor 8](#_Toc508968475)

[Commands 8](#_Toc508968476)

[Base 8](#_Toc508968477)

[CircleCrop 8](#_Toc508968478)

[Convolution 8](#_Toc508968479)

[CustomCrop 9](#_Toc508968480)

[EllipseCrop 9](#_Toc508968481)

[Filter 9](#_Toc508968482)

[GaussianBlur 10](#_Toc508968483)

[LimitColors 10](#_Toc508968484)

[Multiplier 10](#_Toc508968485)

[Negative 10](#_Toc508968486)

[SquareCrop 10](#_Toc508968487)

[Vignette 10](#_Toc508968488)

[Creating Custom Commands 11](#_Toc508968489)

[Step 1 – New Script 11](#_Toc508968490)

[Step 2 – Imports 11](#_Toc508968491)

[Step 3 – Inheritance 11](#_Toc508968492)

[Step 4 – Implement GetPixel 11](#_Toc508968493)

[Step 5 – Check File 11](#_Toc508968494)

[Step 6 – Write Algorithm 11](#_Toc508968495)

[Step 6.1 – Simple Algorithm 12](#_Toc508968496)

[Step 6.2 – Complex Algorithm (but still simple) 13](#_Toc508968497)

[Creating Custom Crop Commands 15](#_Toc508968498)

[Step 1 – New Script 15](#_Toc508968499)

[Step 2 – Imports 15](#_Toc508968500)

[Step 3 – Inheritance 15](#_Toc508968501)

[Step 4 – Implement keepPixel 15](#_Toc508968502)

[Step 5 – Check File 15](#_Toc508968503)

[Step 6 – Write Algorithm 15](#_Toc508968504)

[Step 7 – Use Command 16](#_Toc508968505)

[Step 8 – Check Image 16](#_Toc508968506)

Photo Editor V2 Docs

Made by Jordan Roth

# Quick Intro

This documentation should be easy to follow and understand. I want to make sure everything is straight to the point, and I will try to cut out a lot of the “bloat” text as much as possible. If you enjoy this asset please leave me a review on the asset store, I would greatly appreciate it.

# How to Contribute (Needs Finished - links)

Since this asset is free means I rely on different ways of keeping this project alive rather than strictly by money. If you love this asset, or believe it needs some work, please consider utilizing any of the following ways to help push this project forward. If you believe this project is worth money (and if you are able) please consider a donation through itch.

## Asset Store (click for link)

Leave a rating

Tell your opinions

## GitHub (click for link)

Give a star

Report issues

Make requests

Submit pull requests

## Itch (click for link)

Give a donation

Leave a rating

Post a comment

# Quick Start

## **Step 1 - Imports**

**Create a new C# script and set up the imports**

using Picture\_Editor\_v2.Scripts; // Contains Texture2DEditor class   
using Picture\_Editor\_v2.Scripts.Commands; // Contains all of the commands

## Step 2 – Global Variables

Make the global variables

public MeshRenderer RendererToManipulate; // Place to store the edited texture  
public Texture2D Tex; // Original texture  
private Texture2DEditor \_myTextureEditorVariable; // Changes (a copy of) the texture

## Step 3 – Start Method

Use the start method to make the edits

void Start ()  
{  
 // Initialize the editor with the texture  
 \_myTextureEditorVariable = new Texture2DEditor(Tex);  
   
 // Add commands in the order they should execute  
 \_myTextureEditorVariable.AddCommand(new CircleCrop());  
 \_myTextureEditorVariable.AddCommand(new GaussianBlur(5));  
   
 // Call "GetTexture2D" to attain the new texture  
 RendererToManipulate.sharedMaterial.mainTexture = \_myTextureEditorVariable.GetTexture2D();  
}

## Step 4 – Check File

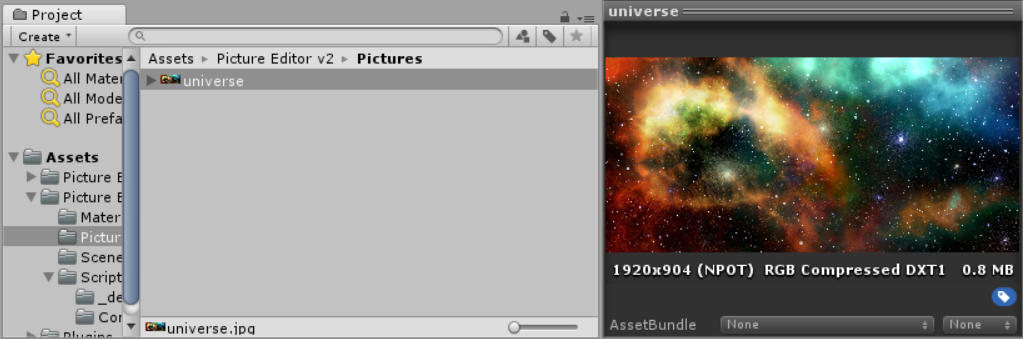
Double check your file looks like the following

using Picture\_Editor\_v2.Scripts; // Contains Texture2DEditor class   
using Picture\_Editor\_v2.Scripts.Commands; // Contains all of the commands  
using UnityEngine;  
  
public class Test : MonoBehaviour  
{  
 public MeshRenderer RendererToManipulate; // Place to store the edited texture  
 public Texture2D Tex; // Original texture  
 private Texture2DEditor \_myTextureEditorVariable; // Changes (a copy of) the texture  
  
 void Start ()  
 {  
 // Initialize the editor with the texture  
 \_myTextureEditorVariable = new Texture2DEditor(Tex);  
   
 // Add commands in the order they should execute  
 \_myTextureEditorVariable.AddCommand(new Filter(Filters.Sepia));  
 \_myTextureEditorVariable.AddCommand(new GaussianBlur(2));  
   
 // Call "GetTexture2D" to attain the new texture  
 RendererToManipulate.sharedMaterial.mainTexture =

\_myTextureEditorVariable.GetTexture2D();  
 }  
}

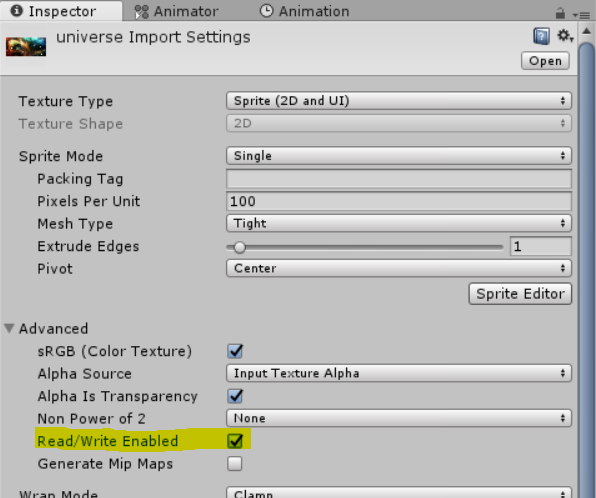
## Step 5 – Find Image

Find the image you want to edit

****

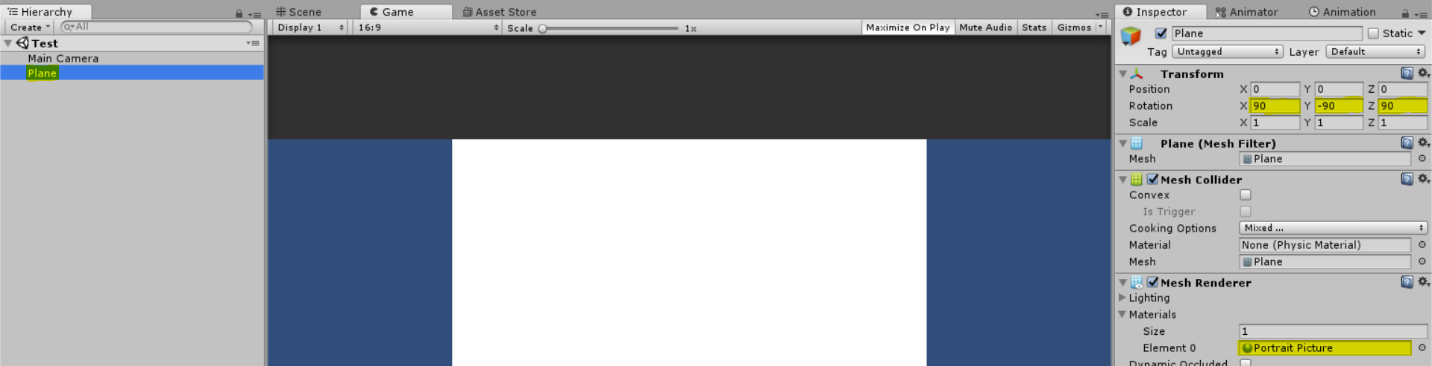
## Step 6 – Image Settings

Enable “Read/Write” on the image (and hit apply)

****

## Step 7 – Add Plane

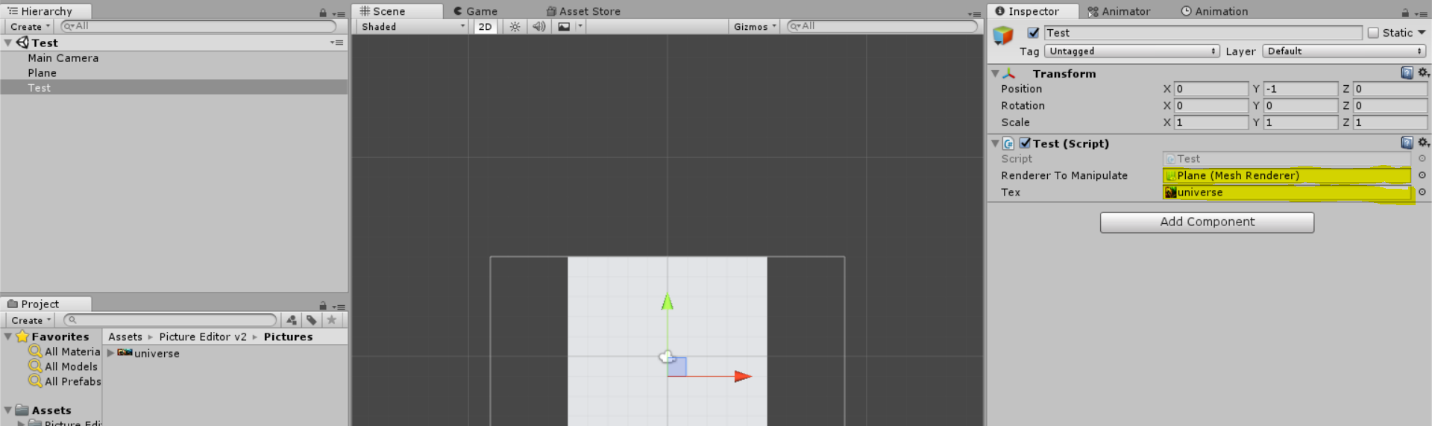
Add a plane object to the scene and set “Portrait Picture” as the material

****

Note: If you have more than 1 object that uses Portrait Picture, then if you edit one of them, you edit all of them. To avoid this, simply create a copy of Portrait Picture and apply it to the other objects.

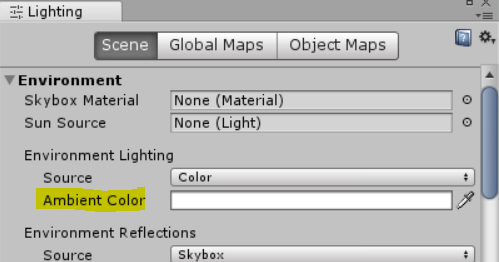
## Step 8 – Add Script

Add a new object with the script you just wrote and setup the variables

****

## Step 9 – Set Lighting

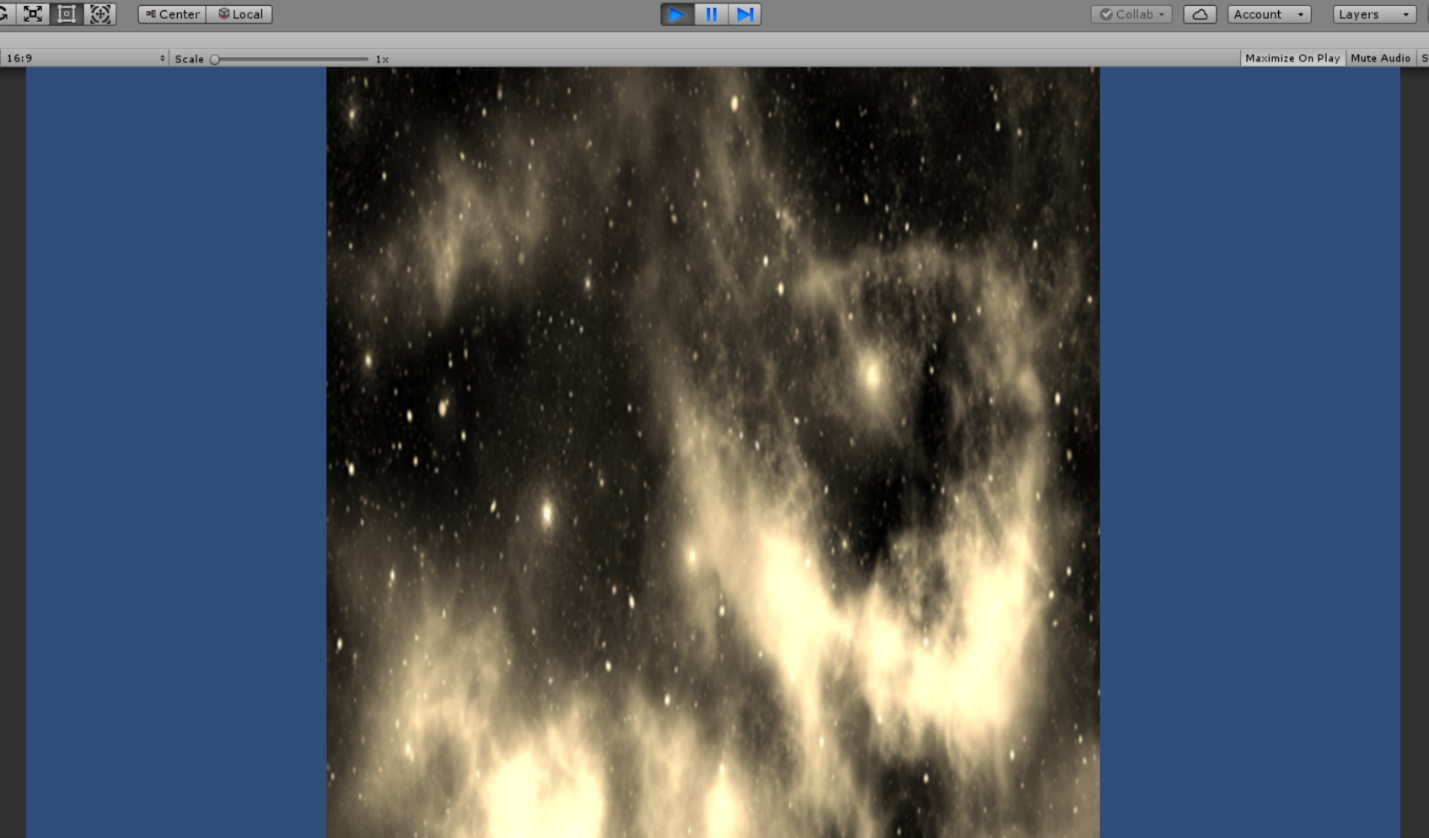
Make sure the lighting is set up correctly

****

The Lighting tab can be found under Window > Lighting > Settings

## Step 10 – Hit Play

Hit “Play” to see the applied edits

****

Note: The reason why it is square is because of the Plane object. You can reshape the scale of the Plane object within the Transform settings of the game object.

# Understanding the Scripts

## Texture2DEditor

// Creates a new Texture2DEditor object (only takes 1 parameter)  
Texture2DEditor myEditor = new Texture2DEditor(tex: editableTexture);  
  
// Adds a command to the editor  
myEditor.AddCommand(command: myCommand);  
  
// Returns the new Texture2D with the edits applied  
// Also trims clear pixels off the sides of the texture  
myEditor.GetTexture2D();  
// Does not trim the pixels  
myEditor.GetTexture2D(trim: false);

## Commands

### Base

// Has no effect, you will probably never need this  
EditCommand myCommand = new Base();

### CircleCrop

// Cuts a circle from the center to the nearest edge  
EditCommand myCommand = new CircleCrop();

### Convolution

// Uses convolutions to edit a pixel based off of surrounding pixels  
EditCommand **myCommand** = new Convolution(new float[3,3]);  
  
// Keeps the edges in the image  
myCommand = new Convolution(Convolutions.EdgeDetection);  
  
// Keeps the edges with their depth  
myCommand = new Convolution(Convolutions.EdgeDetectionDepth);  
  
// Makes the edges more apparent   
myCommand = new Convolution(Convolutions.EdgeEnhancement);  
  
// Embosses the image (it looks like you pushed into it)  
myCommand = new Convolution(Convolutions.Emboss, 0.5f);  
  
// Embosses the image giving attention to pixels at 45 degrees  
myCommand = new Convolution(Convolutions.Emboss45, 0.5f);  
  
// Embosses the image with depth  
myCommand = new Convolution(Convolutions.EmbossDepth, 0.5f);  
  
// You guessed it, intensely embosses the image  
myCommand = new Convolution(Convolutions.EmbossIntense, 0.5f);  
  
// Adds more contrast to the image  
myCommand = new Convolution(Convolutions.Sharpen);  
  
// Greatly contrasts the image (perhaps too much)  
myCommand = new Convolution(Convolutions.SharpenIntense);  
  
// Mediumly contrasts the image  
**myCommand** = new Convolution(Convolutions.SharpenMedium);

### CustomCrop

// Stretches a black and white image and only keeps the pixels  
// in the same location as the white pixels  
EditCommand myCommand = new CustomCrop(blackAndWhiteImage);

### EllipseCrop

// Crops out an ellipse from the image, leaving  
// everything outside of the ellipse clear  
EditCommand myCommand = new EllipseCrop();

### Filter

// Keeps percentages of colors at a certain pixel  
EditCommand **myCommand** = new Filter(new float[3,3]);  
  
// Keeps percentages of colors at a certain pixel with alpha  
myCommand = new Filter(new float[4,4]);  
  
// Only keeps the blue in each pixel  
myCommand = new Filter(Filters.Blue);  
  
// Only keeps the blue and green in each pixel  
myCommand = new Filter(Filters.Cyan);  
  
// Creates a gray scale image  
myCommand = new Filter(Filters.Gray);  
  
// Only keeps the green in each pixel  
myCommand = new Filter(Filters.Green);  
  
// The english spelling of the gray filter  
myCommand = new Filter(Filters.Grey);  
  
// Only keeps the red and blue in each pixel  
myCommand = new Filter(Filters.Magenta);  
  
// Only keeps the red in each pixel  
myCommand = new Filter(Filters.Red);  
  
// Weirdly switches around the colors (ex. the value in red  
// may become the value in blue -- also 0 can be replaced with  
// any value in the range from 0 to 4)  
myCommand = new Filter(Filters.RotateHue0);  
  
// Colors become more vibrant  
myCommand = new Filter(Filters.Saturate);  
  
// Makes the image look "older"  
myCommand = new Filter(Filters.Sepia);  
  
// Gives the image a yellow look  
**myCommand** = new Filter(Filters.Yellow);

### GaussianBlur

// Blurs the image, the higher the number, the higher the intensity  
EditCommand myCommand = new GaussianBlur(3);

### LimitColors

// Limits each color to n colors. You may just have to play around to  
// see what happens, but when n is set to 3, there will be a total  
// of 9 colors that the image can be.  
EditCommand myCommand = new LimitColors(3);

### Multiplier

// Lightens the image when the value is less than 1 and  
// darkens when greater than 1. (1 has no effect)  
EditCommand myCommand = new Multiplier(2f);

### Negative

// Negates the image  
EditCommand myCommand = new Negative();

### SquareCrop

// From the center to the nearest edge, crops out a square  
EditCommand myCommand = new SquareCrop();

### Vignette

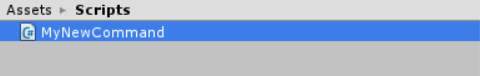
// Creates a vignette around the image, starting at begin percent to the nearest edge,  
// and ending at end percent to the nearest edge.  
//  
// The backwards variable is a bit experimental, and I don't see many uses for it, but  
// it inverses the vignette  
EditCommand myCommand = new Vignette(begin: 0.5f, end: 1f, backwards: false);

# Creating Custom Commands

For those who wish to make a command compatible with the Texture2DEditor class

## Step 1 – New Script

Make a new script



## Step 2 – Imports

Set up the imports

using Picture\_Editor\_v2.Scripts.Commands;  
using UnityEngine;

## Step 3 – Inheritance

Inherit from the EditCommand class

public class MyNewCommand : EditCommand { … }

## Step 4 – Implement GetPixel

GetPixel is the instructions of how your code will manipulate each individual pixel (more on this later)

public override Color GetPixel(int pos)  
{  
 throw new System.**NotImplementedException**();  
}

## Step 5 – Check File

Make sure your file looks something like the following

using Picture\_Editor\_v2.Scripts.Commands;  
using UnityEngine;  
  
public class MyNewCommand : EditCommand {  
  
 public override Color GetPixel(int pos)  
 {  
 throw new System.**NotImplementedException**();  
 }  
   
}

## Step 6 – Write Algorithm

Right now, there are two ways that you may need to set up your Algorithm. Either you need to know the width and/or height for calculations you need to make before your algorithm, or you don’t. Usually you will need the height and or width to find the center of the image beforehand.

### Step 6.1 – Simple Algorithm

If you don’t need to know the width and/or height of the image for anything that can be calculated beforehand, you may write all of the algorithm in the GetPixel function

#### Step 6.1.1 – Write Algorithm

Let’s say you want to write an algorithm that applies a green filter, as well as makes the green value lighter, then you could write your GetPixel method like the following

using Picture\_Editor\_v2.Scripts.Commands;  
using UnityEngine;  
  
public class MyNewCommand : EditCommand {  
  
 public override Color GetPixel(int pos)  
 {  
 // Get the previous color  
 Color **color** = \_getPixel(pos);  
   
 // Set the color to only the green value  
 **color**.r = 0;  
 **color**.b = 0;  
   
 // Make the green lighter  
 **color**.g = **color**.g \* 1.2f;  
   
 // Clamp the color since the green value may be larger than 1  
 ClampColor(ref **color**);  
  
 // Finally, return the newly constructed color  
 return **color**;  
 }  
   
}

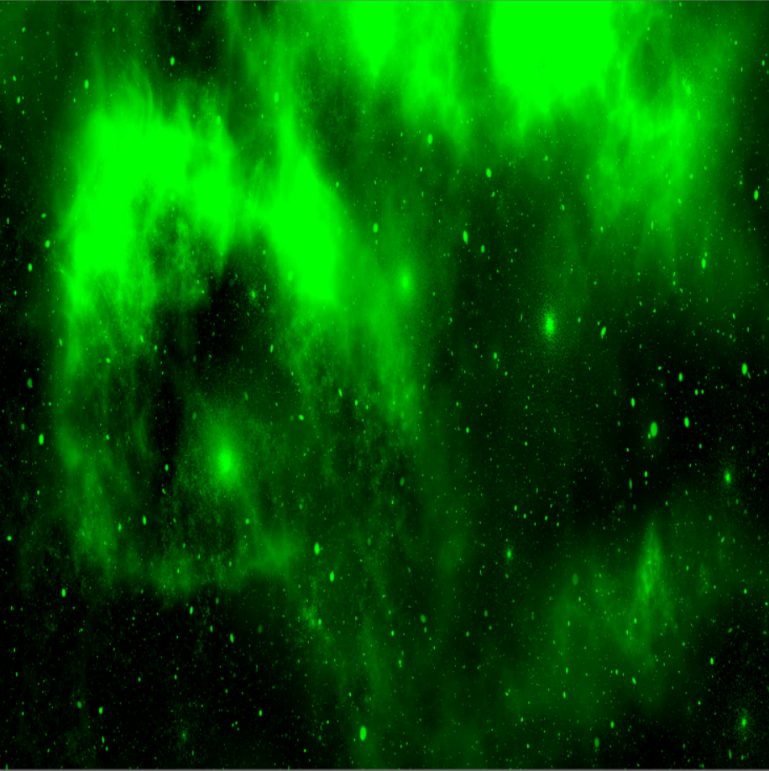
#### Step 6.1.2 – Use Command

Assuming everything went correctly, you should be able to produce the following code in your script that controls your image (see quick start step 4)

Texture2DEditor myEditor = new Texture2DEditor(Tex);  
  
EditCommand myCommand = new MyNewCommand();  
myEditor.AddCommand(command: myCommand);  
  
RendererToManipulate.sharedMaterial.mainTexture = myEditor.GetTexture2D();

#### Step 6.1.3 – Check Image

Make sure your image looks light green



### Step 6.2 – Complex Algorithm (but still simple)

If you need the width and/or height of the image, then you will need to set up one extra method

#### Step 6.2.1 – Write Algorithm

In contrast to the simple algorithm, let’s say now you want to keep a larger fraction of the green closer to the middle, and a larger fraction of red and blue closer to the edges, then your MyNewCommand class could resemble the following

using Picture\_Editor\_v2.Scripts.Commands;  
using UnityEngine;  
  
public class MyNewCommand : EditCommand  
{  
  
 private int \_centerX;  
  
 public override Color GetPixel(int pos)  
 {  
 // Get the previous color  
 Color **color** = \_getPixel(pos);  
   
 // Find the x position  
 int xPosition = pos % \_texture2DColor.Width;  
 // Find how far the x position is from the center.  
 // Notice that \_centerX is precalculated, saving a little  
 // bit of time with each iteration  
 int distanceFromCenter = Mathf.Abs(xPosition - \_centerX);  
 // Reduce the distance range between 0 to 1  
 float multiplier = distanceFromCenter / (float) \_texture2DColor.Width;  
   
 // Remove green from edges  
 **color**.r \*= multiplier;  
 **color**.b \*= multiplier;  
 **color**.g \*= 1 - multiplier;  
   
 // No need to clamp the colors since there is no chance that  
 // any colors will be outside of the range 0 to 1, but it doesn't  
 // hurt too much to be safe  
 ClampColor(ref **color**);  
  
 // Finally, return the newly constructed color  
 return **color**;  
 }  
  
 // Use initialize to calculate the center x position beforehand  
 public override void Initialize()  
 {  
 \_centerX = \_texture2DColor.Width / 2;  
 }  
}

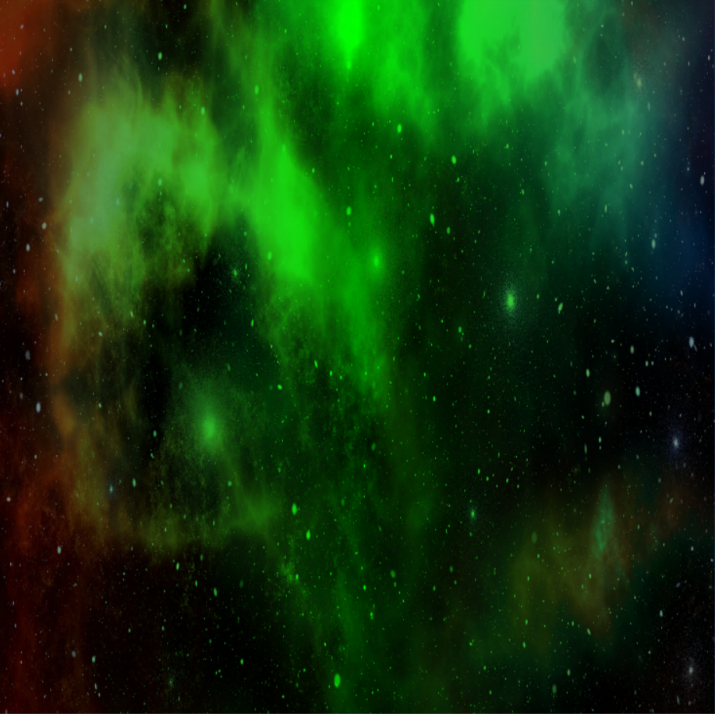
#### Step 6.2.2 – Use Command

This is the same as 6.1.2: Assuming everything went correctly, you should be able to produce the following code in your script that controls your image (see quick start step 4)

Texture2DEditor myEditor = new Texture2DEditor(Tex);  
  
EditCommand myCommand = new MyNewCommand();  
myEditor.AddCommand(command: myCommand);  
  
RendererToManipulate.sharedMaterial.mainTexture = myEditor.GetTexture2D();

#### Step 6.2.3 – Check Image

Make sure your image looks green in the middle and without green on the edges

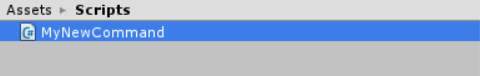


# Creating Custom Crop Commands

A simple way to make algorithms for custom cropping

## Step 1 – New Script

Make a new script



## Step 2 – Imports

Set up the imports

using Picture\_Editor\_v2.Scripts.Commands;

## Step 3 – Inheritance

Inherit from the Crop class

public class MyNewCommand : Crop { … }

## Step 4 – Implement keepPixel

keepPixel will return true if the pixel at the position should be kept, and false otherwise

protected override bool keepPixel(int pos)  
{  
 throw new System.**NotImplementedException**();  
}

## Step 5 – Check File

Make sure your file looks something like the following

using Picture\_Editor\_v2.Scripts.Commands;  
using UnityEngine;  
  
public class MyNewCommand : EditCommand {  
  
 public override Color GetPixel(int pos)  
 {  
 throw new System.**NotImplementedException**();  
 }  
   
}

## Step 6 – Write Algorithm

For the sake of simplicity, let’s say you want to keep every other pixel (maybe for a see-through effect), then it should look like the following

using Picture\_Editor\_v2.Scripts.Commands;  
  
public class MyNewCommand : Crop  
{  
 protected override bool keepPixel(int pos)  
 {  
 return pos % 2 == 0;  
 }  
}

## Step 7 – Use Command

Assuming everything went correctly, you should be able to produce the following code in your script that controls your image (see quick start step 4)

Texture2DEditor myEditor = new Texture2DEditor(Tex);  
  
EditCommand myCommand = new MyNewCommand();  
myEditor.AddCommand(command: myCommand);  
  
RendererToManipulate.sharedMaterial.mainTexture = myEditor.GetTexture2D();

## Step 8 – Check Image

Make sure your image looks see-through (my background is blue)

