# [Article] Painting on 3D terrain

Years ago, I played an RTS where the *borders between countries* where clearly painted on the terrain, and shifting constantly.

I was both impressed and dumbfounded.

How could they do this? How could they draw all these complex lines and have them stick perfectly to this huge terrain?

Over the years, I saw this technique in many other games, and had the same question: how?

Well, today, when I was actually doing something else, I realized I missed the obvious solution for years. So let me explain how to do it in this short article :)

I’ll be using **Godot Engine** as the example, but it should work similarly in any other engine.

## Step 0: Draping cloth

This is where I went wrong with my thinking: I thought that these games *painted the lines on the actual texture, at exactly the right size and locations*. That seemed an impossible task, only doable with endless math and shader trickery!

But that’s not what they were doing. (At least, in most cases.)

Instead, you must look at it this way. We’re going to:

* Create a big flat 3D plane that covers the whole terrain. (Or whatever you want to paint on.)
* Then let it *fall down* so it *drapes over the terrain*, like a piece of cloth.
* Now we have a plane that matches the height differences of the terrain, which we can paint by simply giving it a texture and adding/removing stuff from it.
* (The actual painting depends on the implementation. You’d probably want to convert the *global position* to the *position local to the plane*, and then update the pixels there.)

## Step 1: Creating the plane

Create a **MeshInstance.** Give it a **PlaneMesh.** Set its width and height to whatever is big enough to cover the whole area you want paintable.

Now turn the **subdivision** numbers up to something quite high. The higher you go, the more *detailed and precise* the painting will be. But it will come at a performance cost.

## Step 2: Letting the vertices fall down

This has three steps:

* Loop through all vertices in the plane.
* Shoot a downward *raycast* into the physics world.
* Reposition the vertex to whatever position it hit.

Here’s how to do it in Godot (GDScript):

TO DO: Code here

But the same steps should be easily reproducible in other engines. Every game engine has great support for raycasts (as they are vital) and most have easy tools for modifying mesh vertices.

## Step 3: Adding the texture

Okay, so we have a mesh that perfectly covers all the hills and nooks of our world. (Or at least *close enough*.)

Now we can give it a *material*/*shader* and input the *texture* that it needs to display.

If you just want, for example, a border around the world, that’s easy. Just draw a texture with a line near the edges and input that.

But we want more. We want to paint anything on the terrain. Maybe entities leave prints when walking around. Maybe you’re making a shooting game and want (realistic) blood spatters all over the environment.

To do so, we must:

* Create an empty image we can hold and manipulate through *code*.
* Paint its pixels whenever something happens, at the right location.
* Convert it to a texture, and hand it to that PlaneMesh, every frame.

Godot uses ImageTexture for handling images through code, which can (at any point) be converted to an actual Texture to give to a material. Other engines might not draw this distinction, or use slightly different words. But again: the idea is the same and they should support it.

TO DO: Code

## Step 4: Painting the texture

The tricky part is the **painting at the right location.**

Shaders and textures work with UV coordinates. The upper left of a texture is (0,0), and the lower right is (1,1). If you want to sample, for example, the exact center of a texture, you’d sample UV coordinate (0.5, 0.5).

This also means shaders start from the *top left*, so we need to account for that.

In conclusion, we need a way to convert the

* *Global position we want painted* …
* *…* to its position *local to the upper left of the mesh*
* … to its *UV coordinates*
* … and then back to its *actual pixels in the texture*.

To make matters worse, we need to decide upon a **resolution**. There’s a limit to how large textures can be (before the game becomes laggy or just plain crashes). If you want to paint a big environment, you need to decide upon a scale: how many *pixels* do I give to each *unit/meter in the 3D world?*

In the game that made me realize this technique, I gave 16 pixels to each “unit”. In other words, the texture size was the *world dimensions* multiplied by *16,* which came in at around (1280,720) pixels. But that was a cartoony game, with relatively small worlds.

The best thing is, of course, to make this a variable you can change at any time.

The calculations for this are greatly simplified if you position your mesh at (0,0,0), the world origin. You *can* put it anywhere and account for that, but why would you do that?

Putting this all together, we get …

At game start:

* Create an ImageTexture …
* … with size <dimensions of plane mesh> \* <resolution>

Whenever something wants to paint on it …

* “Hey, something happened at position X, paint something there!”
* Local position = X + <half dimensions of plane mesh>
* UV coordinates = local position / (full dimensions of plane mesh)
* Pixel coordinates = UV coordinates \* full image size = Y
* Now set the pixels at Y in the image to the color you want.

If you want to paint more complex things, like circles, other textures, etcetera, you can either:

* Use a built-in function to “stamp” those into the texture (at Y)
* Or use some *loops* or *mathematical equations* to calculate which pixels to paint (around Y)

I often draw circles, which means I simply loop through a rectangle around the point Y, and paint any pixels within a certain distance ( = circle radius).

There’s some overhead to drawing individual pixels, but it gives you the most customizability: you can change color, shape, distances, alpha, etcetera on the fly.

TO DO: Code

## Step 5: That’s it!

You have a mesh sitting on top of your environment. You can convert any location to its correct position in the *image* given to the mesh.

Now you can do anything you want, and it will show up *at the right location*, with the right *height/shape/normal*.

Of course, it’s not perfect. Some detail is lost, some things might get stretched out a bit. But it’s more than good enough, and relatively cheap, to achieve this effect.

Here are some last things to watch out for:

**Z-fighting:** When repositioning the vertices to sit on top of the terrain, add a *tiny* value in the Y-axis to actually make it on top. Otherwise it will be at the exact coordinates as the terrain, which creates Z-fighting: both meshes are at an identical location, so the computer constantly (randomly) chooses which one to display, causing it to flicker.

**Wrong mesh:** It’s vital that the plane is evenly subdivided, and generously subdivided. Any other shape complicates matters by a lot. Cheaping out on the subdivision will not help you at all: raycasting and vertices are *cheap* for todays hardware, so don’t hold back!

**Use it multiple times or in smaller ways:** Of course, there’s no need to keep this technique to one *humongous* texture overlaying the whole world. You can easily break this up into smaller chunks. Maybe you have smaller pieces draped over areas that need it more. Maybe you only paint the terrain in the immediate surroundings of the player, or when some weapon is active.

I can’t tell you how happy I was when I realized this is how you do it. It’s not hard, it’s not so tough on the computer, yet it gives great control and “feel” to a game world to actually *permanently interact with your environment*.

It’s funny how things that seem impossible at first, become simple after some more years of experience and the right Eureka at the right time.

(Of course, I often searched for the solution to this problem. But either nobody else is talking about it, or I searched in the wrong direction. I could only ever find some clues, or vague hints, or *similar* techniques discussed online. For example, there’s a lot on how to do it in 2D, but it doesn’t really translate to 3D.)

Hopefully this helps your next game project. If anything is unclear, or you’re trying to do something more advanced, don’t hesitate to contact me and maybe I can help!

Until the next time,

Pandaqi