CSCI 444/544 - Lab 4
Friday, February 12, 2021
LAB IS DUE BY **FRIDAY FEBRUARY 19 11:59 PM**!!

This lab will introduce you to 2-Dimensional noise.

Step 0 – A New Library File

Copy the file in include/CSCI441 to your folder. This is a hastily thrown together UBO wrapper class to assist with the usage.

Great! Run the program, you should see a white quad.

Step 1 – Take A Look Around

The uniforms are being sent via UBO to provide a working example. The second UBO is a bit more important and more technical for its need. GLSL has a thing against arrays. We can't define them when we declare them, there's no constructor. (If interested a short related description http://math.hws.edu/graphicsbook/c6/s3.html#webgl.3.2).

From reading the description of the Perlin Noise algorithm, we need to store either the grid of gradients in a 2D array or use a permutation table for our hashing to coordinates. We're going to use the second approach since GLSL only supports 1D arrays (though OpenGL 4.3 has an extension to support 2D arrays).

The array length is 512 values so we do not want to manually assign each index its value in the shader. This computation would be done in the fragment shader and would then be redone every fragment every frame. Instead, we'll precompute it CPU side and send it via uniform once to the GPU.

The precomputed array in `main.cpp`, in the variable `p` and then `pTable`. In `setupShaders()`, we send over the integer array just once. Take a look at how this is done. When mixing UBOs and arrays, GLSL is very particular. All components within a uniform block array get expanded to be four words. Meaning, a single int is stored in the X component for a ivec4 (an integer vec4) even though the GLSL only references a single int. Therefore, we need to pack our data to match this expansion. On the CPU, we convert our integer array to a padded ivec4 array. It is this ivec4 array that gets put in to the UBO.

And that's it on the OpenGL side, everything happens in shader land.

Step 2 – Send through the varying

When we get to the noise calculation, we need a coordinate to compute the noise at. We will ultimately use the interpolated vertex coordinate.

To do this, create a `vec2` varying from the vertex shader to the fragment shader. Assign its value in the vertex shader to be the XZ components of our vertex coordinate. Our plane lies in the XZ plane, so we will use the corresponding reference point.

Step 3 – I can't hear you, it's too noisy

Now everything else goes in the fragment shader.

We'll create some helper functions to assist in the calculations. Refer to Ken himself for assistance in Java on how to implement each function https://mrl.nyu.edu/~perlin/noise/ImprovedNoise2D.java .

The first is to compute the `fade` of a parameter. The function should take a single `float` as input, return a single `float` as output, and compute the following equation:

$$6t^5 - 15t^4 + 10t^3$$

The next function computes our pseudorandom gradient. Convert Ken's `grad()` function in Java to the representation in GLSL. Be sure to work with `floats` instead of `doubles`. This is Ken's simplification to computing the dot product.

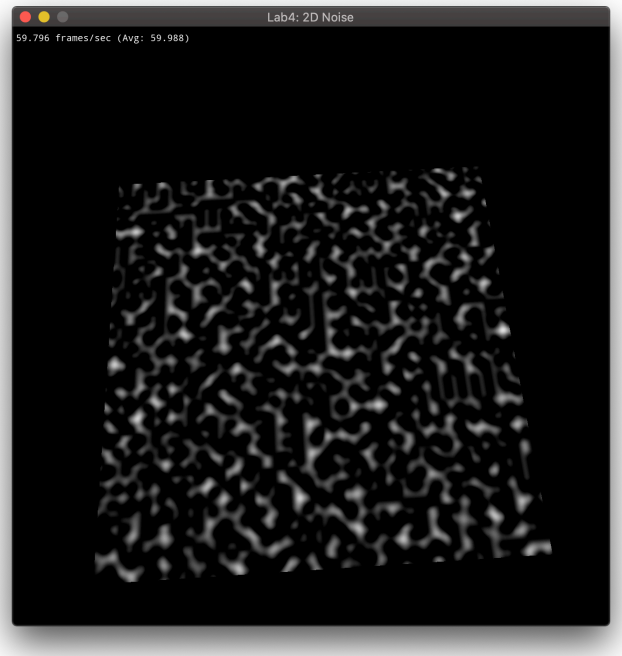
We don't need Ken's `lerp` function, GLSL has a built in `mix` function to do it for us. Do not use `lerp()` in your solution – for full credit, convert his solution to use the GLSL `mix()` function. *(Hint: the parameters are in a different order.)*

Now all that is left is to create a `noise` function. It accepts two `floats` as input and returns a single `float` as output. His comments describe the steps of the algorithm to implement:

1. The first two lines find the unit square that the point is in
2. The next two lines find the fractional component of our point to know the relative point inside the square
3. The next two lines compute the fade of each parameter to smooth the transitions
4. The next two lines find the hash of the four coordinates of the unit square
5. The final four lines blend the results of the four coordinate dot products

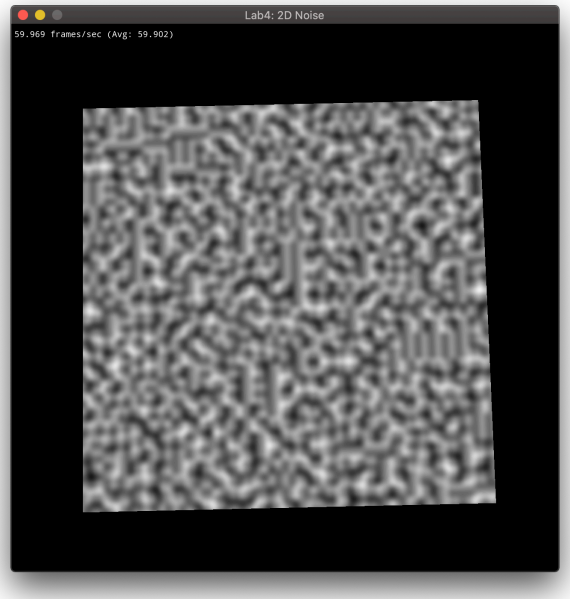
Once we have created the `noise` function, we'll now call it in our fragment shader's main passing it the varying value we received as input. Store the result of the function as a variable, we'll call it `n`.

Now set our `fragColorOut` to have the `RGB` components all equal to `n`. Run the program and you should see an image like the following:



Our noise function is generating values between `[-1, 1]` so half the image is below `0`. Prior to setting the `fragColorOut`, modify `n` by shifting it up by one and normalizing the result to modify the range to be `[0, 1]`.

Rerunning should give us a brighter image with some more detail:

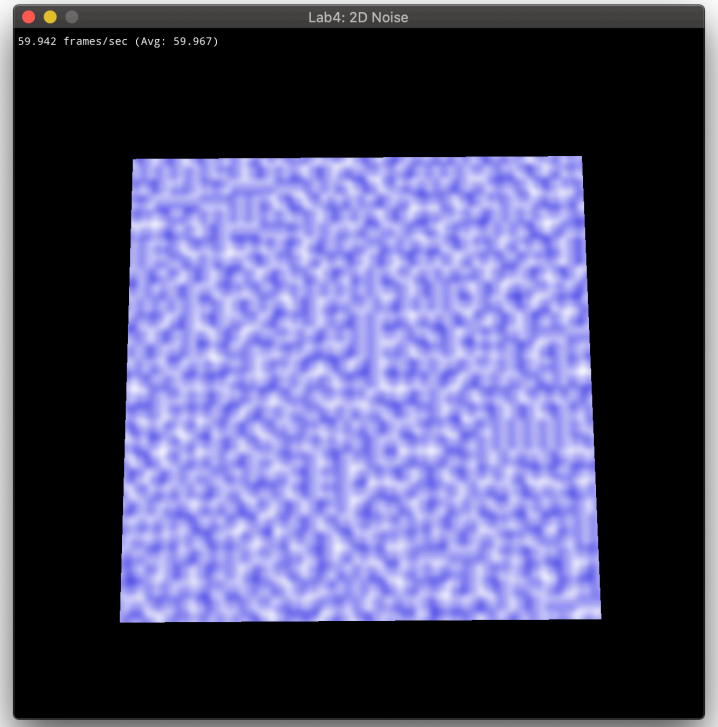


Let's make it cloudy now. Right now we are just displaying our normalized noise value. Let's use that parameter n to interpolate between two colors – the sky and the cloud.

Create two colors to represent the `skyColor` and the `cloudColor`. I'll recommend `(0.3, 0.3, 0.9)` for the sky and `(1.0, 1.0, 1.0)` for the clouds. But you can use different colors to create different moods – a pretty sunset, a dark and stormy night, etc.

We'll no longer set the `fragColorOut` to be just a gray scale version of our noise. We'll now `mix` between the `sky` and `cloud` using our normalized `n` parameter.

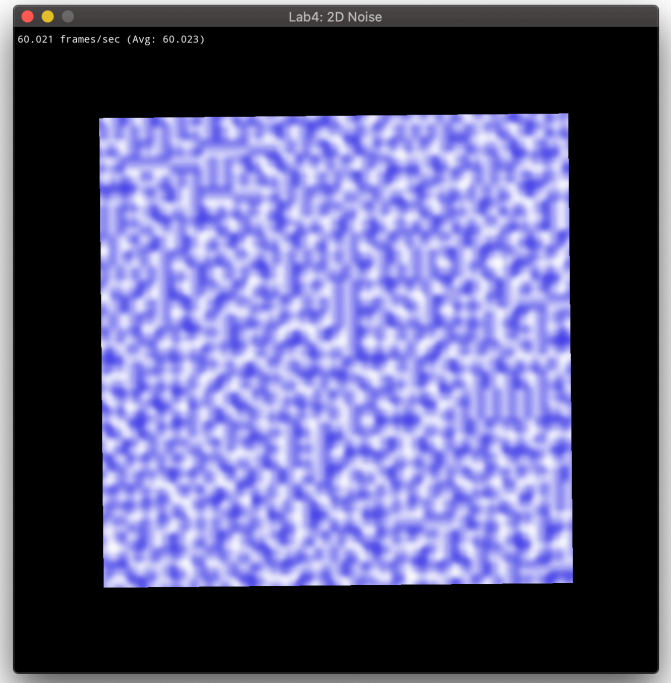
Now, we're starting to see some color!



We still have some knobs to turn and dials to twist. Let's first smooth the transitions by applying cosine to our noise parameter. Modify `n` a second time by setting it equal to the following equation

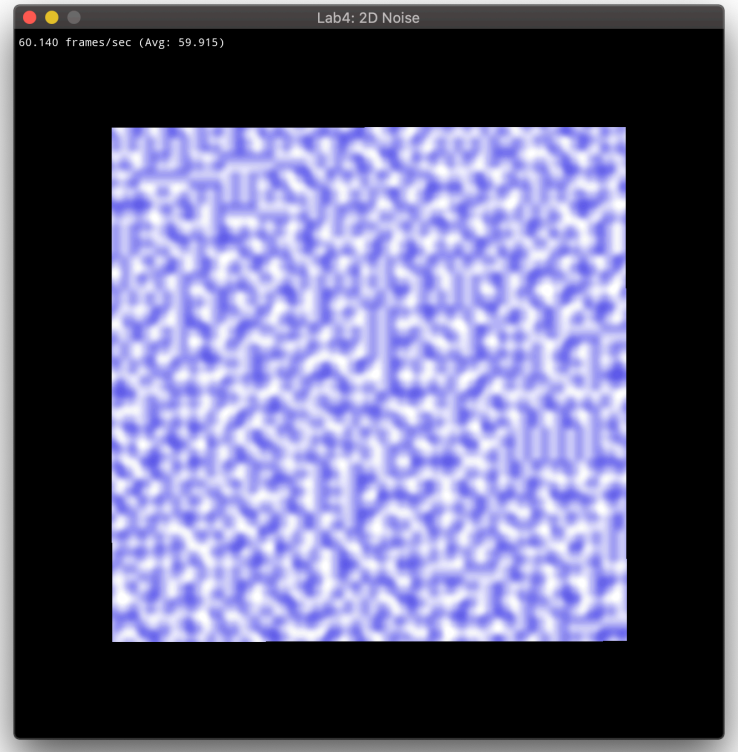
$$\frac{cos(n\pi) + 1.0}{2.0}$$

We should now have crisper boundaries between sky and cloud.



The last value we can modify is how much cloud there should be relative to the sky – so called the `cloudLevel`. We'll modify `n` by adding the `cloudLevel` (try 0.1 to start) and then clamping (see `clamp()`) the result to the range `[0, 1]`.

We now have the resultant image!



To recap the steps just done:

1. Generate noise value `n`
2. Normalize from `[-1, 1]` to `[0, 1]`
3. Smooth by `cos(nπ)`
4. Normalize from `[-1, 1]` to `[0, 1]`
5. Shift phase by `cloudLevel` and `clamp` to `[0, 1]`

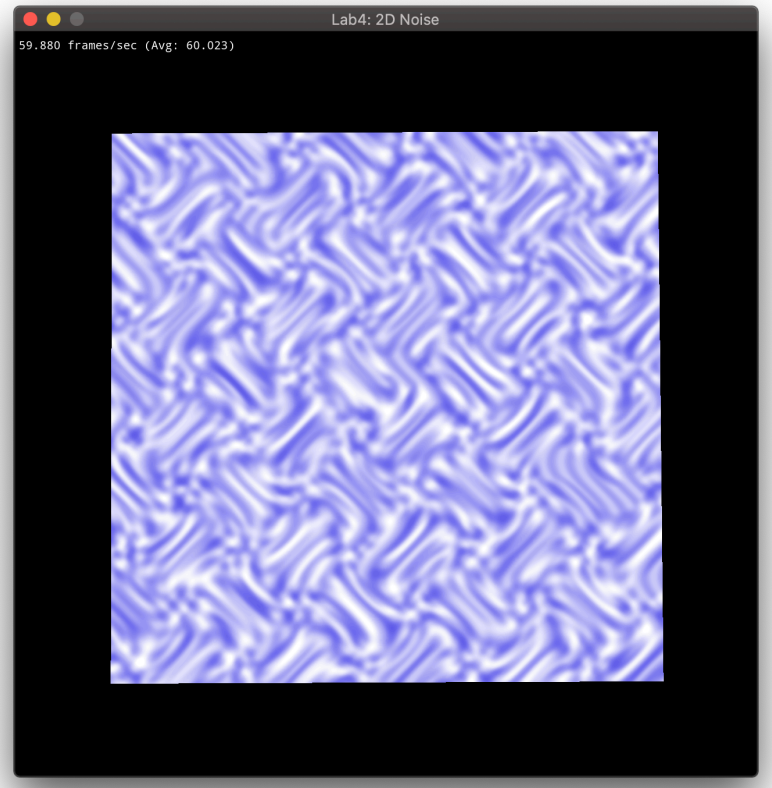
Step 4 – This one goes to 11

Let's see what effect changing inputs to the noise function has on the result. We have everything set up to do so. In the vertex shader, prior to sending out our varying variable, we'll add in the time component. Modify the `X` value by adding `cos(time)` and modify the `Z` value by adding `sin(time)`.

Rerun and you'll notice the texture moving in a circular pattern.

Let's mess with the coordinates even more. In the fragment shader, before sending the coordinate to our noise function, add `cos(time + t)` to the S coordinate and add `sin(time + s)` to the T coordinate.

Rerun and we now get a much different result.



Step 5 – What's Next?

As far as this lab, we're done. Submit your project source code to Canvas as a single zip file.

Despite our noise "moving", it is still only 2-dimensional noise. We are only shifting the phase of where we lie in the plane.

For the next assignment, our noise will be expanded to work in 3- and 4- dimensions to apply the procedural texture to our teapot and have vary through time. Now our texture will be moving through space and changing. To produce more visually pleasing results, multiple octaves will be added together.

LAB IS DUE BY **FRIDAY FEBRUARY 19 11:59 PM**!!