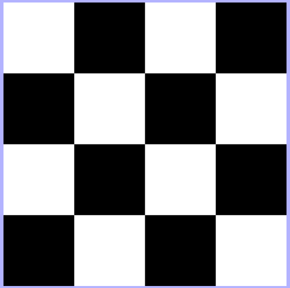
More Textures

Motivation

- We've seen basic texturing
- Now we'll examine more advanced texturing strategies

Recall

- Suppose we have a square with texture coordinates that go from 0...1
- And suppose we have a checkerboard texture
- We might get something like this:

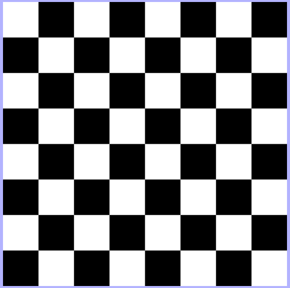


Question

- All our texture coordinates were in range 0...1
- What if they go outside that range?
- Alter our code: Make texture coordinates go from 0...2
  tbuff = Buffer(array.array("f", [0,0, 0,2, 2,2, 0,2]))

Result

- The texture repeats itself:

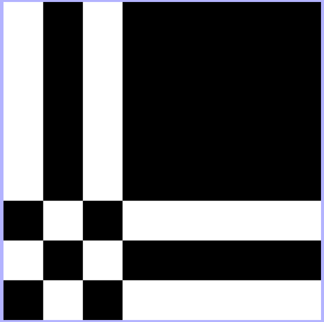


Repeat

- Whether we get repeat is governed by the sampler object
- Change the sampler constructor:
  glSamplerParameteri(self.samp, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE)
  glSamplerParameteri(self.samp, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE)

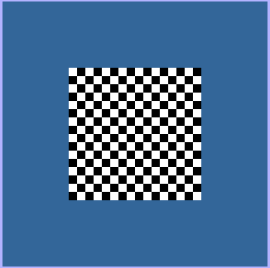
Result

- Anywhere the coordinates would have been less than 0 or greater than 1: get *clamped*
- Remember: (0,0) is lower left corner



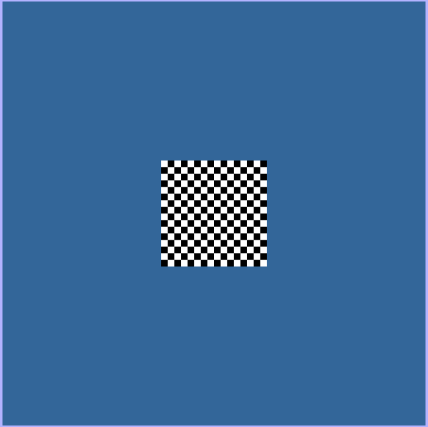
Now

- Suppose we change the position values so they aren't -1...1 anymore
- Maybe try -0.5...0.5
  - And while we're at it, make texture coords go 0...4
  - And turn repeat back on: Use GL_REPEAT instead of GL_CLAMP_TO_EDGE in sampler setup
- ```
  p=0.5
  vbuff = Buffer( array.array("f", [ -p,-p, p,-p, p,p, -p,p ]) )
  t=4
  tbuff = Buffer( array.array( "f", [ 0,0, 0,t, t,t, 0,t ] ) )
  ```


  ```

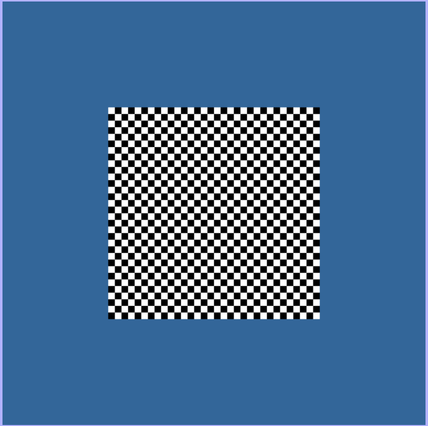
And

- Now make p=0.25



So...

- Let's re-set the positions to be -0.5...0.5 and make the texture coordinates go 0...8

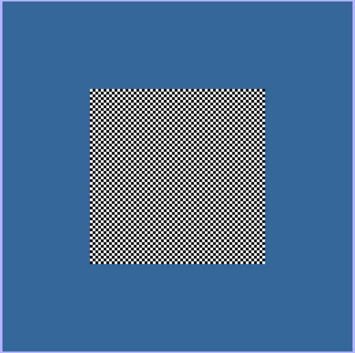


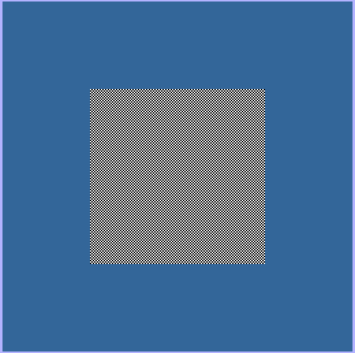
Notice

- If we make the object smaller, the checks get smaller
- Or, if we make the texture coordinates larger, the checks get smaller
- Maybe it’s obvious when you think about it...

Now

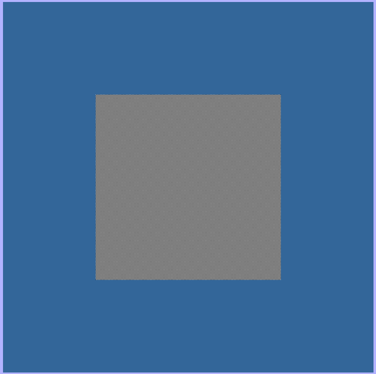
- Suppose we bump t to 16. And then to 32.





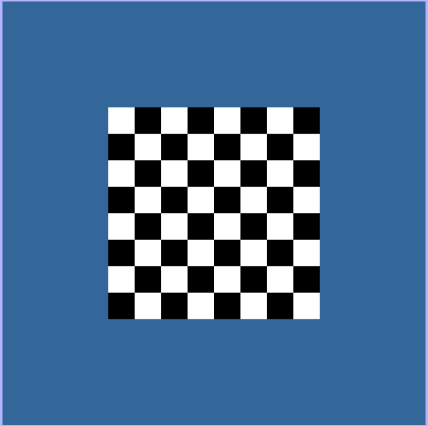
Well...

- It seems the limit would be to set t=64
- Then we have one black pixel, one white pixel, one black pixel, ...



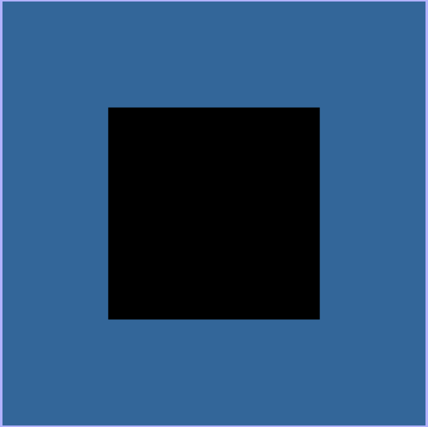
Question

- What happens if we set t > 64?
- How about t=130?



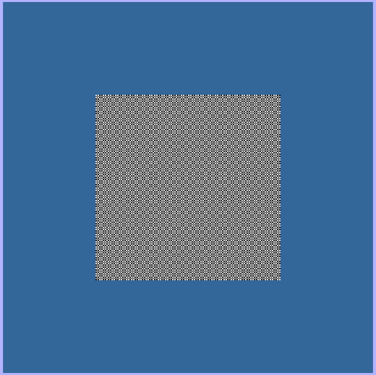
Hmm...

- What if t=128?



Or...

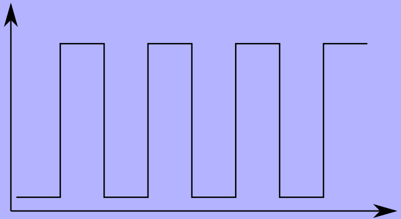
- Maybe t=90?

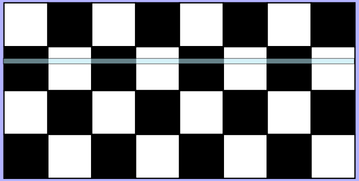


- What is going on?

Explanation

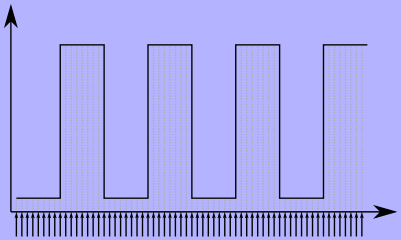
- Consider single scan line.
- Its colors alternate from black to white.
- We can graph intensities (y axis) versus position (x axis)

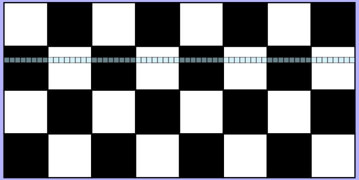




Explanation

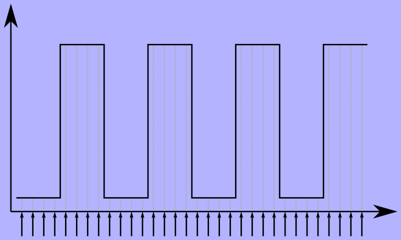
- Camera is close to object (or texture coordinates' range is small) $\rightarrow$ Dense sampling
- Image looks OK

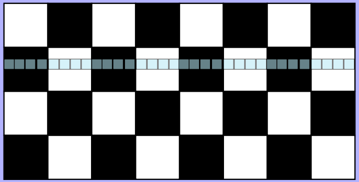




Explanation

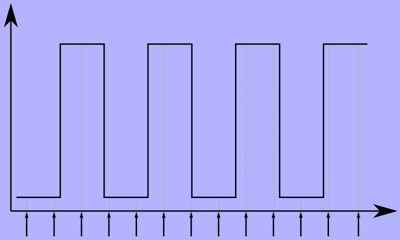
- Camera moves further away (or texture coordinates' range gets larger) $\rightarrow$ Not as dense sampling
- Reconstruction still OK.

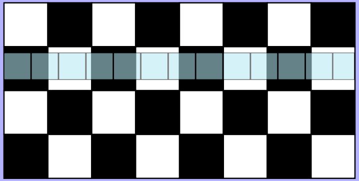




Explanation

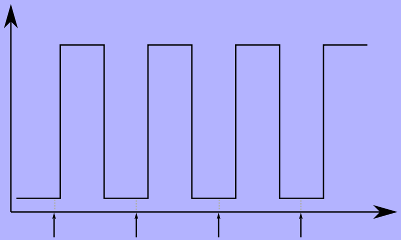
- Camera is further away (or texture coordinates' range gets very large) $\rightarrow$ Sparse sampling
- Checks are uneven now

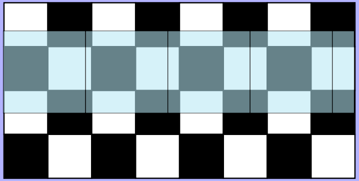




Explanation

- Camera is even further away (or texture coordinates' range very very large) $\rightarrow$ Very sparse sampling
- "Large check" pattern
- This is *aliasing*
  - Name comes from signal theory





Human Eyes

- Two factors help prevent aliasing in human eyes:
  - Photoreceptors are not in regular grid pattern
    - Regular patterns make aliasing more prominent
  - Cornea acts as “low pass filter”
    - Blurs out high frequency components

Solution

- We can’t really change the grid pattern of monitor pixels
- But we can simulate the cornea’s action
  - The further something is, the more we blur it

How?

- How to accomplish blurring?
  - Too time consuming to do on the fly
  - How can we quantify amount of blur to perform?

Mip Mapping

- Mip mapping: Introduced in 1983 by Lance Williams
  - Mip = Multum in Parvo = Many things in a small place
- Preprocess the texture to get successively more blurred versions of it
- Store them all
- At runtime, select one based on distance from viewer
  - Further away = More blurred
- The details are a bit subtle...

Question

- How to determine distance from viewer?
- GPU uses a heuristic: For each 2x2 quad of pixels:
  - Look at texel coordinates accessed for each quad
  - Difference between them allows us to estimate size of a pixel in texel space
- The bigger the pixel is in texel space, the further away the object must be

MIP

- Bigger pixel size $\rightarrow$ Biases the selection to a more blurred copy of image
- Also referred to as a *higher mip level*

Terminology

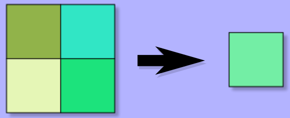
- Mip level 0 = the original, full detail image
- Mip level 1 = a little blurred
- Mip level 2 = more blurred
- etc.

Blurred Images

- One problem: Storing many copies of image chews up GPU memory
  - We will likely have *many* textures and models competing for space
  - So we don't want to waste RAM needlessly

Solution

- Suppose we have mip level 0: Our original image. Suppose it has size 128x48
  - Take four adjacent pixels from mip level 0 and average them together to get one pixel
  - Do this for all the pixels of level 0
  - This gives us an image that is half the size of level 0 (64x24)
  - We'll refer to this as mip level 1



Solution

- Repeat this process to get mip level 2 (32x12)
- And again for level 3 (16x6)
- And once more for level 4 (8x3)
- One more time for level 5 (4x1)
  - Notice: 3/2 gives a noninteger quotient
  - Power-of-two dimensions work best for mipmapping, but modern GPU's can handle NPOT textures
- Level 6 will be 2x1
- And level 7 is 1x1. We're done.

Memory

- We said memory usage was a concern.
- How much RAM does a mipmapped image require?
- Suppose input is 32x32. RAM: 32×32*times;3 = 3072 bytes
- Level 1 = 16x16x3 = 768
- Level 2 = 8x8x3 = 192
- Level 3 = 4x4x3 = 48
- Level 4 = 2x2x3 = 12
- Level 5 = 1x1x3 = 3
- Total: 4096 bytes
  - Which is 133% of original 3072 bytes
- Mip mapping adds overhead of 33%.

Result

- How do we use mipmapping?
- GL includes it built-in
- We must request it from our sampler: Change MAG filter to GL_LINEAR and MIN filter to GL_LINEAR_MIPMAP_LINEAR

And

- In ImageTexture2DArray: After the glTexImage3D call: glGenerateMipmap(GL_TEXTURE_2D_ARRAY)
- This is important: Without this, the mip chain is *incomplete*
- GL will always return black (0,0,0,0) for texture with incomplete mip chain
- My GPU gives a debug message:
  ```
  131204 : Texture state usage warning: The texture object (1)
  bound to texture image unit 0 does not have a complete set of
  mipmaps and cannot be used with a sampler needing mipmaps.
  ```

Result

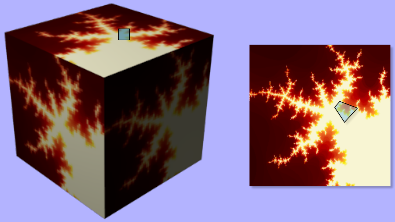
- With t=90: The checkers are starting to blur out to grey

Result

- With t=130: Notice it's now a uniform grey color, simulating a blurred checkerboard

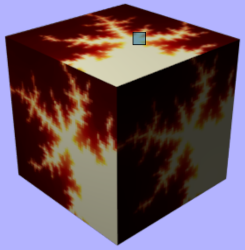
Texel Shape

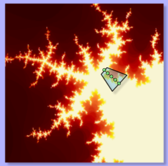
- Recall how we said we'd select mip level: Look at difference between texture coordinates to estimate pixel size in texture space
- But: A pixel is usually *not* a square in texture space
  - What mipmap level to use? s says one thing, t says another
  - Could average them, prefer maximum one, prefer minimum one, etc.



Anisotropic

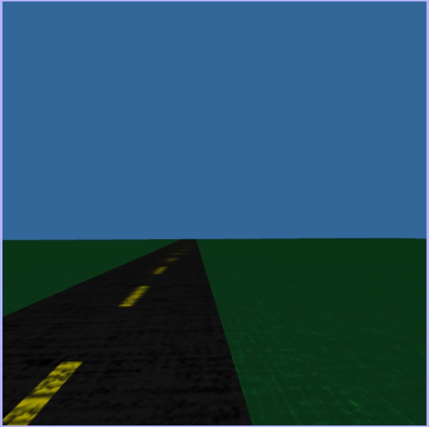
- Another strategy:
  - Map four corners of pixel to texture space
  - Take several samples along longest axis
  - Average them together
- Reduces blurring, especially in perspective views
- Slower (but has hardware support)

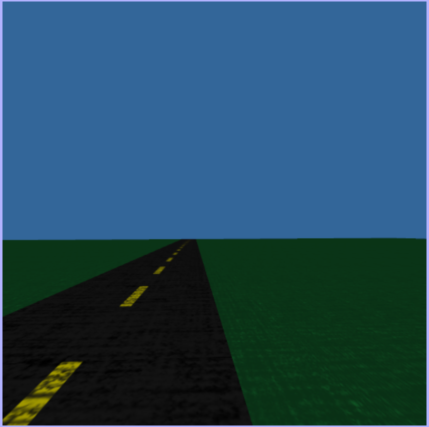




Anisotropic

- Left: Linear mipmap filtering
- Right: Anisotropic filtering





To Use

- To use anisotropic filtering: Set GL_TEXTURE_MAX_ANISOTROPY_EXT sampler parameter
  - Value is an integer: 1=no anisotropy, 2=use 2 samples, 3=use 3 samples, ...

Assignment

- Enable mipmaps
- Add enemies that spawn at random times and move right-to-left across the screen. When an enemy goes off the left side, it should be deleted
- Your enemy should have a unique texture.
- For bonus [+25], have two types of enemies (with different textures): One which moves right-to-left and the other drops down from above.

Sources

- A. Watt & M. Watt. *Advanced Animation and Rendering Techniques.* Addison-Wesley.
- Sampler Object. OpenGL Wiki. https://www.khronos.org/opengl/wiki/Sampler_Object

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org