TEXTURING

Texture

Textures are array of data, much like other buffer objects, however there are important distinction between them in terms of data access and storage allocation. We access (index) texture data in a fragment shader through texture coordinates which are real numbers between 0 and 1, also known as UV coordinates. Texture access further depends on dimensionality of texture: 1D texture could be indexed by a real number between 0 and 1, 2D texture could be indexed by pair of real numbers and likewise for 3D textures. The data itself may have upto 4 components. 1D, 2D, 3D textures conceptually form a grid of data. The smallest discrete "pixel" in this pixel grid is texel.

We allocate storage for texture in OpenGL by calling one of glTexImage1D, glTexImage2D or glTexImage3D. These apis require us to specify format of our input, and the format we would like GPU to store it internally, also known as internal format.

This internal format e.g. GL_R8 specifies:

1. No. of components stored in a texel.
2. No. of bits each component take.
3. Data type of component.

This behavior is different from glBufferData api which only requires us to provide the format of input data, and the internal format of that data is specified later through glVertexAttribPointer. Even then, texture storage is still different than buffer storage since GPU may internally rearrange texture data, while this is not the case with buffer data.

Normalized Integers

Normalized Integers are memory efficient way of storing floating point values on range [0, 1] with fewer than 32 bit integers. For e.g. it is a common optimization to map floating point values into [0, 255] 8 bit unsigned integers. To map this integer back to floating point value, we simply divide by 255.

OpenGL assumes components of texture data, by default to be normalized unsigned integer. For e.g.: Having GL_R8 as internal format in glTexImage2D call mean components to be stored as unsigned integers internally. In contrast, we would have to specify GL_R32F, to have components stored as 32 bit floating point numbers.

GLSL Sampler

GLSL Sampler are how we reference a texture inside shader. Just like 1, 2 or 3D texture, we have corresponding sampler1D, sampler2D and sampler3D types in shader. The apis around samplers are like opaque pointers in C, in that these samplers are passed as argument to different OpenGL functions. Indeed samplers have interesting restrictions:

- Sampler can only be declared at global scope as uniform or in function parameter with the 'in' qualifier.
- Samplers cannot be member of struct or uniform block.
- Only use of sampler types are as arguments to functions. User defined functions can list them as parameters, and there are number of built in functions that take sampler.

Here is how we declare a sampler globally for 1D texture type:

```
uniform sampler1D myTextureSampler;
```

Built in 'texture' function accepts sampler as first argument and texture coordinate as second.

For 1D texture, texture coordinate is a real number between [0, 1] and refer to texel coordinate [0, texture size].

To sample or fetch a texel from middle of texture, we would call

```
float f = texture(myTextureSampler, 0.5).r;
```

Note that built in texture function returns Vec4 regardless of texture format. Therefore, '.r' above extracts first component of that Vec4, which is the texel value for 1D texture.

Texture Binding

We need to associate the texture we uploaded with glTexImage2D call and the sampler inside shader.

OpenGL has array of slots called texture units. We bind a texture to a texture unit; we likewise set association between sampler variable and that texture unit. This associates texture with the sampler variable inside shader.

Setting sampler uniform variable to texture unit:

```
GLUint textureUnit = 3;
GLUint samplerRef = glGetUniformLocation(
                        program, "myTextureSampler");
glUseProgram(program);
glUniform1i(samplerRef, textureUnit);
```

Setting texture object to texture unit:

```
GLUint textureUnit = 3;
GLUint textureObject;
glActiveTexture(textureUnit);
glBindTexture(GL_TEXTURE_1D, textureObject);
```

Texture Coordinate Interpolation

We usually specify texture coordinate as vertex attribute. Say we specify texture coordinates over a vertices that appear rectangle in camera space. After perspective projection, that rectangle might get skewed in following manner:

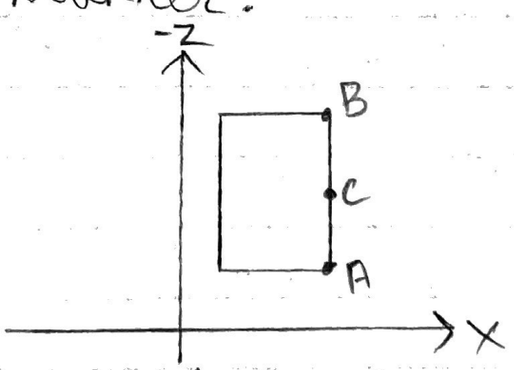

fig: Camera Space

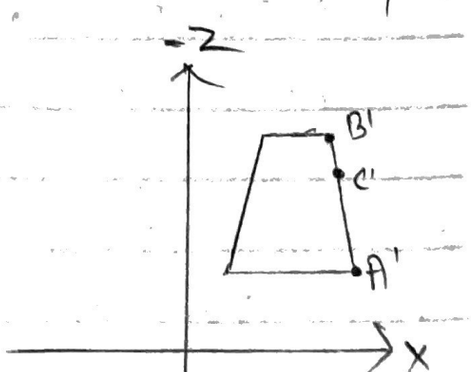

Fig: NDC Space

Notice how C, that is a mid point between A and B in camera space is skewed towards B' in NDC space and likewise in window coordinate. A perspective correct interpolation would interpolate the vertex attribute of this C' in NDC space as if it were C in camera space.

That is, for a texture coordinate vertex attribute

$$\text{TextureCoordinate\_C'} = \text{TextureCoordinate\_C} = \frac{\text{textureCoordinate A} + \text{textureCoordinate B}}{2}$$

This perspective correct interpolation is also known as interpolation in space that is linear to camera space. OpenGL does this by default when we qualify our vertex attribute output with 'smooth' (or no qualifier at all) in vertex shader.

OpenGL 2D Texture Data format versus Common Image format

2D textureData is written in OpenGL by writing first row (y=0), then second row (y=1) and so on.

Most Image formats however have their first row as ymax, second row as ymax-1 and so on.

This is because Image format use top-left orientation in contrast to OpenGL bottom left.

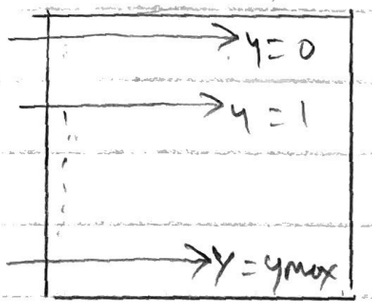

fig: OpenGL 2D texture Data

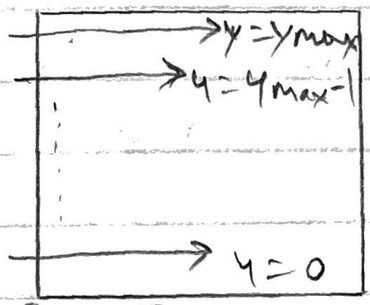

fig: Top-Left oriented Image formats.

Texture Filtering

Consider a 2D texture 128 x 128 in size, and we attempt to access texel at 0.145, 0.145 texture coordinate. This texture coordinate maps to 18.56, 18.56 texel coordinate. Texel coordinates are discrete locations and 18.56 is not. The shader resolves this texture access by either:

Picking the nearest texel i.e. texel at 18, 18 texel location

or,

Averaging 4 surrounding texels.

The first way of resolving is called nearest filtering and second is called linear filtering.

OpenGL additionally distinguishes between magnification and minification filtering. Magnification happens when textures appear bigger in screen than its actual resolution. Minification on the other hand is when texture is shrunk relative to its natural resolution.

Texture filtering for minification and magnification can be specified seperately:

```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER,
                GL_NEAREST);
```

MipMaps and MipMap Filtering

When objects i.e. fragment area that the primitive cover are smaller, then texture mapping be comes inaccurate:

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

fig: Large Texture

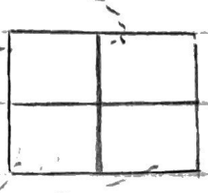

fig: Small Object

In figure above, we see how the object samples small portion of the texture, and if the texture were a picture, the object would not appear right. Instead more reasonable mapping would be sampling down texture to almost the size of the object and then perform texture mapping.

Fig: Minified Texture

Fig: Small Object

Here, we have shrunk our texture (Texture Minification), by averaging out texels from larger texture. Now we see that texture mapping is much aligned.

A mipmap is that smaller version of the texture. In a more technically accurate language, we say that a texture contains many images, with the base image being the biggest and subsequent images being smaller version of that image (mipmaps).

A mipmap image at mipmap level 'n' is twice smaller in every dimension than the one at level n-1. So, a 64X64 original texture image, with mipmaps form a pyramid like

shape with 1x1 mipmap image at top.

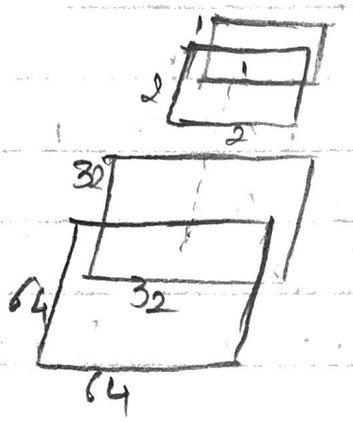

fig: MipMap Pyramid.

Mipmap filtering selects appropriate mipmap before applying texture filtering. Mipmap filtering only happens when objects are smaller relative to textures.

To enable mipmap filtering, provide multiple mipmaps to texture, with glTexImage2D call and then pair GL_MIN_FILTER with one of:

GL_LINEAR_MIPMAP_LINEAR
GL_LINEAR_MIPMAP_NEAREST
GL_NEAREST_MIPMAP_LINEAR
GL_NEAREST_MIPMAP_NEAREST

GL_LINEAR_MIPMAP_NEAREST Filter means the following:

For a GLSL 'texture' function call, it will select mipmap with size that is nearest to our object size (fragment area),

then does linear filtering over the nearest 4 texels as described earlier.

How Mipmap Selection Works

GPU guarantees 4 rectangular blocks of fragment from a primitive, executing in lock step. That means a fragment shader execution not only knows about its interpolated vertex attribute, but also its neighbors. Since texture coordinates are likewise interpolated vertex attribute, every fragment can determine change in texture coordinates w.r.t its neighboring fragments. If u, v are 2D interpolated fragment coordinates, du, dv are the gradient.

du, dv are then multiplied by the width and height of base image and larger of the 2 results is selected. Say, this value is 'α'. Then the mipmap to be selected i.e. mipmap level is given by:

$$mipmapLevel = floor(Log_2(\alpha));$$

$$mipmapLevel = Max(mipmapLevel, 0);$$

Note that for texture magnification i.e.

when textures are stretched to fit to fragment area, $\log_2(\lambda)$ term is always less than 0. So, mipmap level, after that 'Max' operation always ends up being 0. For texture minification with nearest mipmap filtering, we can illustrate with an example:

Mipmap at level 0 is of size 128 X 64

$du, dv = .15, .2$; then,

$\lambda = Max(du * 128, dv * 64)$

$= Max(19.2, 12.8)$

$= 19.2$

Mipmap level $= Floor(\log_2(19.2))$

$= 4$

or 8 X 4 mipmap is filtered.

OpenGL Texture Data Byte Alignment

OpenGL expects texture data to have byte alignment of 4 bytes. That is to say, if we are uploading texture of size 'N', then OpenGL expects 'N' to be divisible by 4. So, even if we specify internal format of GL_RGB, OpenGL expects 4 bytes for every texel.

This byte alignment is default however, and can be changed with glPixelStorei function call.

```
GLVint oldAlign = 0;
glGetIntegerv(GL_UNPACK_ALIGNMENT,
              &oldAlign);
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

Above OpenGL function calls, first save current byte alignment then change the current byte alignment to 1.

SRGB Colorspace

SRGB Colorspace is a non-linear colorspace, where the color values have already been gamma corrected. When an artist exports image from an image editing program, the color values in that picture are most likely in SRGB colorspace. Being non-linear, half brightness value between 0 and 1 scale in SRGB would be somewhere around 0.73.

SRGB and Mipmap

Generating mipmap from a texture involves linear operation like averaging, that ass

color to be in linear colorspace. If the texture were in sRGB colorspace and mipmap were generated by such linear operation we would have incorrect color values. For instance, averaging 0 and 1 in sRGB colorspace we expect the result to be 0.73 but what we would have is 0.5. So, correct way of generating sRGB mipmaps would be to first convert the texture to a linear colorspace, perform linear operations, generate mipmap and finally convert that mipmap back to RGB colorspace.

Note that textures and texture coordinates may represent something other than coordinates and color values; for instance, it may serve as lookup table for some complex computation. In that case, generating mipmaps may not make sense.

sRGB screen.

OpenGL allows us to have our screen image (the default frame buffer) that we initialize with OpenGL context, to already be in sRGB colorspace and the linear color values output via fragment shader to automatically be converted to sRGB colorspace. This gives

us gamma correction for free. To enable this is a 2 step process:

First we specify our implementation specific window / OpenGL context creation function call that we want the default frame buffer to be in sRGB space.

Second, we pass in GL_FRAMEBUFFER_SRGB flag to glEnable, that converts fragment shader output to be gamma corrected.

Note that in cases when we have color values already gamma corrected i.e. in sRGB colorspace, we can disable GL_FRAMEBUFFER_SRGB flag to prevent double gamma correction.