09: Texturing

Ressources

WebGL(2) references
https://www.khronos.org/webgl/
https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
https://webgl2fundamentals.org/

From Webgl fundamentals on texturing
https://webglfundamentals.org/webgl/lessons/webgl-3d-textures.html
https://webglfundamentals.org/webgl/lessons/webgl-2-textures.html

Handin:

A ZIP file containing the following folders and files, **exactly structured like this:**

- lab09-groupXX.zip
    - lab09-groupXX
        - **All files** and folders from the exercise.

***MAC**: To compress to Zip, select a folder in the Finder, right-click and choose "compress". You have to **Zip a folder containing the folder labXX-groupXX** for this to be structured properly:*

- *[AnyFolder] ←- CHOOSE COMPRESS HERE, then name .zip appropriately*
    - *labXX-groupXX*
        - *lab folders and files*

***Windows**: Select the respective folder labXX-groupXX in the Explorer, right-click and choose Send To -> Compressed(zipped) Folder*

Any submission not following these guidelines automatically receives 0 points.

Setup

Create a new folder for the lab. Download the exercise file from
Canvas (Lab 09) and extract all files into it.

Open the lab09-texturing folder. You will find a version of the engine that has been extended to feature the first things we need for texturing, but a lot is still missing. Explore the following files:

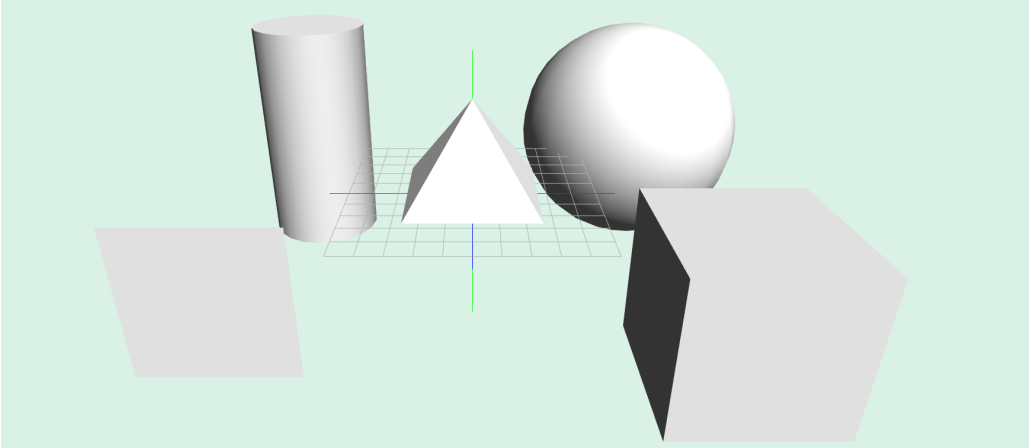
- `engine.html`
  - In the beginning of the file, right below all the imports you can see a new global cache called `TextureCache`. This is a collection of all textures we will be loading and referencing by name. It works similar to the `MeshCache`.
- `primitives.js`
  - Since we now need texture coordinate data, all primitives have a new data array called `texcoords`. It holds the mesh's UV data.
- `common/gl-utils.js`
  - The `createMesh()` function now receives a data array `texcoordData` for the texture coordinates that is used to set up another vertex array buffer: `texcoordBuffer`.
  - To load the textures we want to use for rendering, gl-utils now hosts a function called `loadTexture(name, imgSrc)`.
    - This can load a texture from use supplied by a url or file path `imgSrc`. It will store it in the new global cache `TextureCache` that works just like the old `MashCache`.
    - The function loads the texture async, since it might take some time (using the `onload` callback). Before loading is finished it uses a dummy blue 1x1 pixel texture as a placeholder, so rendering can already begin while loading.
    - The textures are stored as 32bit RGBA format.
    - `gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);` flips the y axis of the image, so it's ready to go for WebGLs texture coordinates.
- `renderer.js`
  - The basic renderer, from here on called Standard Renderer is already capable of rendering tint color and basic directional lighting.
- `shaders/standardShader.js`
  - The basic shader, from here on called Standard Shader is like the Standard Renderer already implementing basic lighting and tint color.

What is the `TextureCache`?

To be able to easily manage all our textures, they are placed in a central collection when loaded: `TextureChace`. When calling `loadTexture(name, imgSrc)` the loaded WebGL Texture Object is given a name `name` (string) and placed in the `TextureChace` collection with that name as an identifier.
To get a texture object from the cache use: `TextureCache[name];`

1: Setting up basic color texturing (50%)

When starting up the program of lab09-texturing, you can see a set of primitives and some basic lighting. The goal is to extend the Standard Shaders functionality to also render a main texture.



To achieve this we need to complete the following steps:

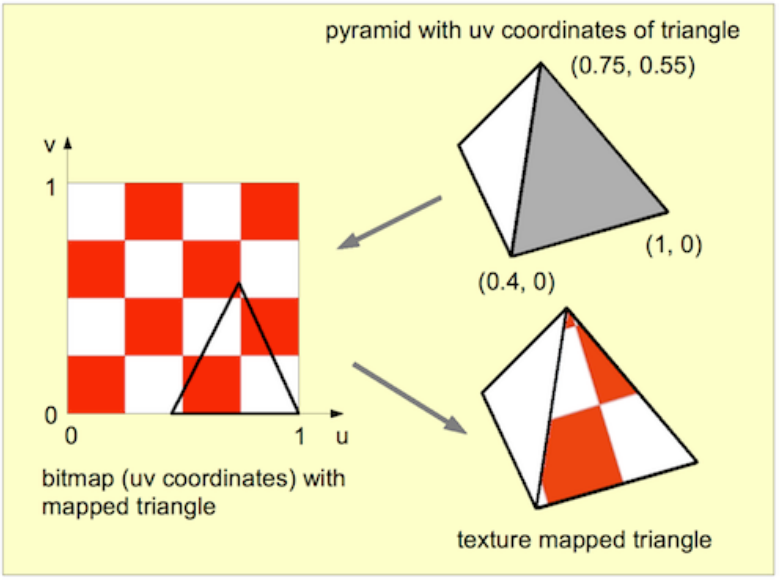
1. Add the texture coordinate attribute.
    a. Add attribute to vertex shader.
    b. Add texture coordinate data and buffer to meshes (mostly already done).
    c. Link texture coordinate buffer and attribute in `renderer.js`.
2. Load an image file as a texture.
3. Make the loaded texture available in the shader for sampling (texture color look-up)
    a. Add a texture sampler to the shader.
    b. Sample the texture (do the texture color look-up based on uv coordinates)
    c. Link the loaded texture to the sampler in `renderer.js`.

Tip: Texture coordinates, texcoords, uv coordinates, uv data or uv's all refer to the same concept.

1.1: Add the texture coordinate attribute

1.1.a: Add attribute to vertex shader.

When using texturing, first we need to make sure our shader can receive and work with texture coordinates (uv coordinates), so our models know which part of the texture has to go where.



- In `shaders/standardShader.glsl` add the `a_texcoord` attribute in the vertex shader.
    - `A_texcoord` has to be a `vec2` since texture coordinates are 2D.

We need the texture coordinates in the fragment shader to do the color lookup. Therefore we have to forward it there using a `varying`.

- Add the corresponding varings in the vertex and fragment shaders.
    - Don't forget to assign the value of the attribute to the varying in the vertex shader.

1.1.b: Add texture coordinate data and buffer to meshes

Of course we need to define the data for our texture coordinates in the meshes. But this is mostly already done for all the primitives. You can see the data arrays in `primitives.js`. and the assignment to the webgl buffer in `common/gl-utils.js -> createMesh(...)`

1.1.c: Link texture coordinate buffer and attribute in `renderer.js`.

- To link up the data from the `texcoordBuffer` of each mesh to our new attribute, complete `setVertexAttributeArrays(...)` in `renderer.js`.

The buffer is stored in `model.mesh.texcoordBuffer`. Remember that uv data is 2D!

1.2 Load an image file as a texture

Loading an image file is quite easy, all you have to do is use the `loadTexture(...)` function found in `common/gl-utils.js`. Textures should be loaded before we start the render loop, so in `init()` in `engine.html`.

We will load the testing texture **uv-test.png** found in the **resources** folder. This texture is quite commonly used to test out if uv coordinates of a model are properly assigned. The grid and number layout gives us easy visual feedback if our uvs are correct.

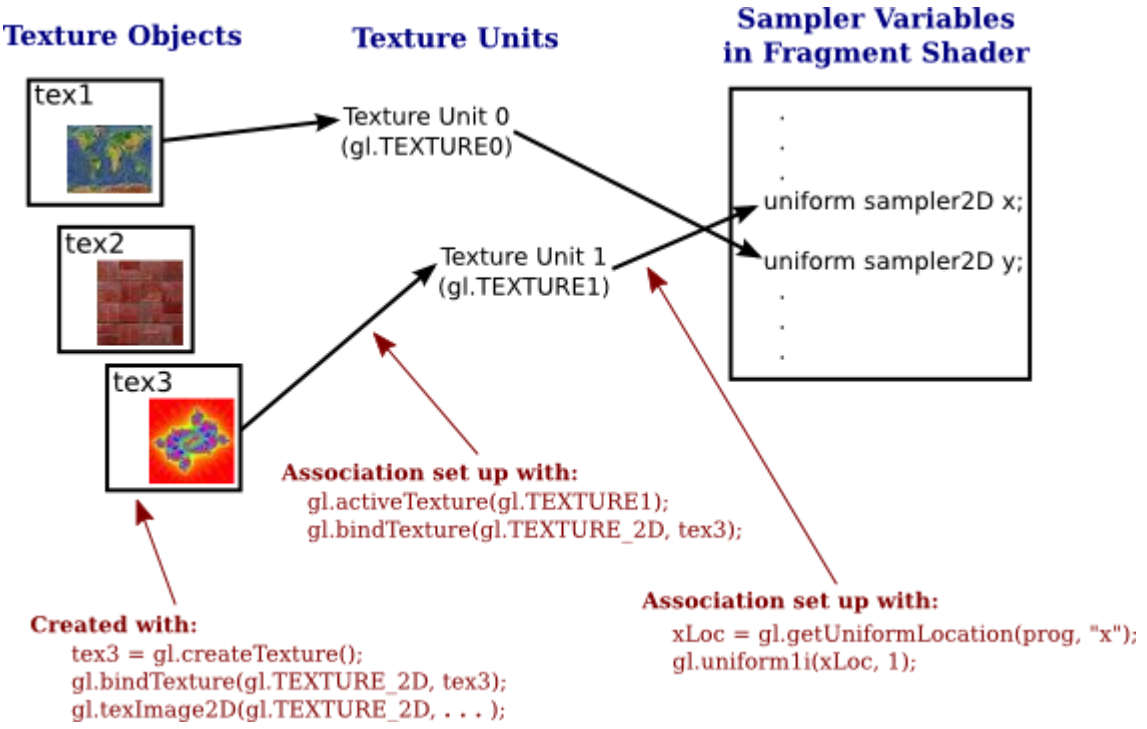
- In `init()` in `engine.html` add:

```
// Load a texture into the TextureCache with the name "uv-test"
GLUtils.loadTexture("uv-test", "ressources/uv-test.png");

// Material with white tint color and uv-test image as main texture
let uvTestMat = {tint: new V3(1,1,1), mainTexture: "uv-test"};
```

1.3 Make the loaded texture available in the shader

This works similar to other uniform data in the shader (see eg. lighting) but textures are bound a bit differently. WebGL works with **Texture Units**, where only a maximum of 5-9 textures can be active at the same time (depending on WebGL version and GPU type).



Our textures are stored in the `TextureCache` as **Texture Objects**. To get them associated with a **sampler** we first need to **bind the Texture Object to a Texture Unit** and then assign the **index of that Texture Unit to the uniform sampler location.**

1.2.a Add a texture sampler to the shader.

We will just work with one main texture for now, that will define the color of our objects. Since textures are treated as uniforms, a sampler `u_mainTex` has to be a uniform.

- In the fragment shader in `shaders/standardShader.glsl` add the texture sampler, the correct syntax and type for this is: `uniform sampler2D u_mainTex;`

1.2.b Sample the main texture

Doing the actual color lookup based on the texture coordinates (sampling the texture) is very easy and just one line of code. The texture color will be blended with our tint color to make the new base color for each pixel of the model.

A texture lookup is done using `texture2D(sampler, coordinates)` in glsl. This returns a `vec4 [r,g,b,a]`. We just want the rgb color as a `vec3`, since we are not using alpha and transparency.

- In the fragment shader in `shaders/standardShader.glsl` add the color lookup from `u_mainTex;`

```
vec3 textureColor = texture2D(u_mainTex, v_texcoord).rgb;
vec3 baseColor = textureColor * u_tint; // blend with tint color
vec3 finalColor = ambientDiffuse * baseColor; // apply lighting to color
```

1.2.c Link the loaded texture to the sampler

When drawing, we always want the texture of the material of the current model we render to be active. This is similar to linking other uniforms, so it happens in `SetUniformData(...)` in `renderer.js`.

Since we only have one texture we will bind it to Texture Unit 0.

- Set Texture Unit 0 to be active:

```
gl.activeTexture(gl.TEXTURE0);
```

To link the Texture Object from our `TextureCache` to the sampler, we first need to retrieve it. The Texture Object is referenced in the material by name: `mainTexture`.
See in `engine.html` in `init()` when we create the material:

```
let uvTestMat = {tint: new V3(1,1,1), mainTexture: "uv-test"};
```

This Texture Object then needs to be bound to the current active Texture Unit.

- Retrieve the Texture Object from the TextureCache using the `mainTexture` reference from the material and bind it as a 2D texture:

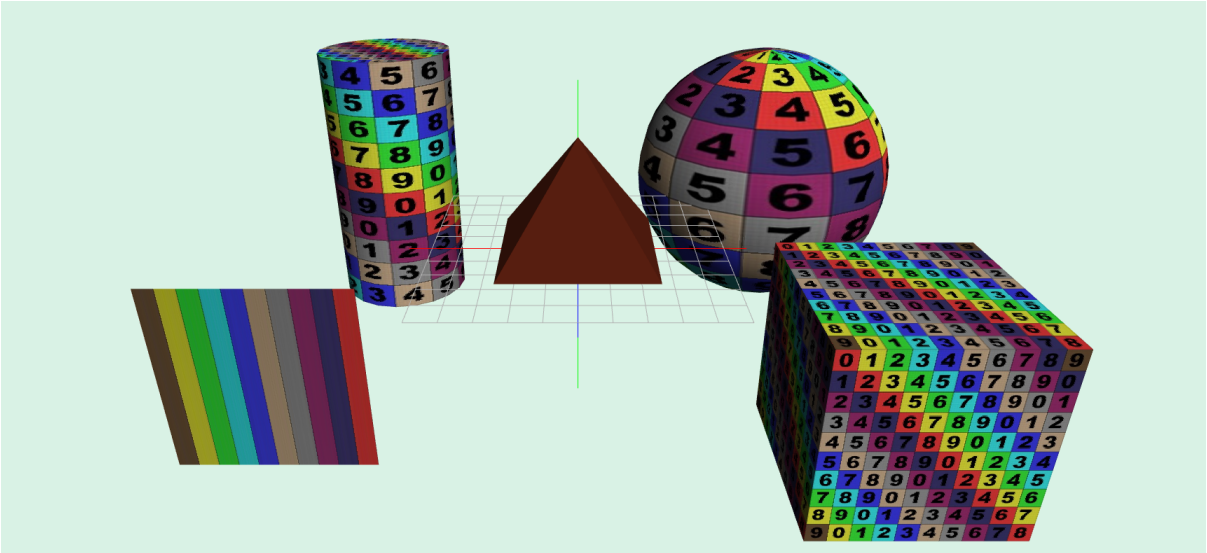
```
let mainTexture = TextureCache[model.material.mainTexture];
gl.bindTexture(gl.TEXTURE_2D, mainTexture);
```

Almost done! Last item on the list is to link up the Texture Unit 0 (now with the Texture Object bound) to the sampler uniform variable in the shader:

- Retrieve the uniform location for the sampler and link it to Texture Unit 0 using the index 0. Use `gl.uniform1i(..)` since the index is 1 integer value.

```
let maintexLoc = gl.getUniformLocation(this.program, "u_mainTex");
gl.uniform1i(maintexLoc, 0);
```

The resulting image should look like this:



There are some obvious problems here that have to be fixed, but more about that in 3.

2. Debugging Texture Samplers and Texture coordinates

If you **do not get the above image**, you have some options to find the error:

2.1 Basic debugging using the Browser Debugger

This is always the first and best option. Use the debugger to inspect the values of your variables in `setUniformData()` and `setVertexAttributeArrays()`. Make sure you get the correct uniform locations and use the `texcoordBuffer`.

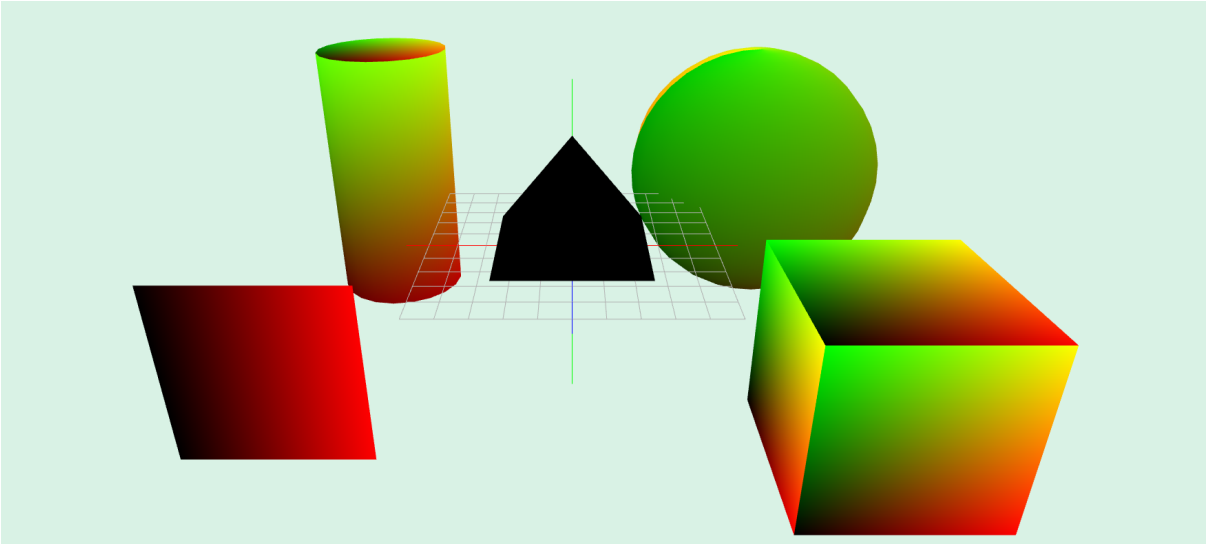
2.2 Debug by color output

Outputting your texture coordinates as a color in the fragment shader is also a good way to find out if your uv data is actually arriving in the fragment shader. This is what we did with the normals in the last lab. Shader debugging primarily works through color output.

Change the last line of your fragment shader to be:

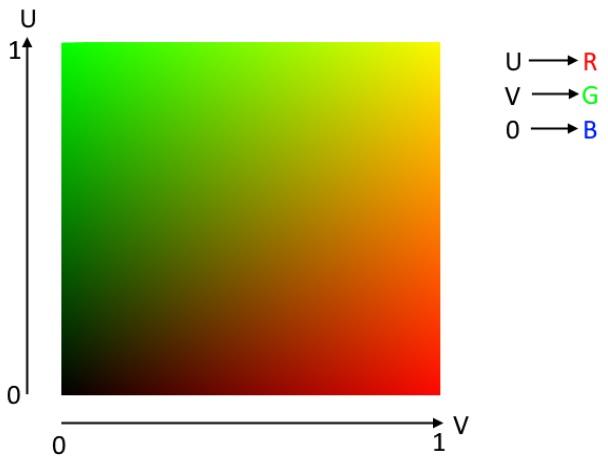
```
gl_FragColor = vec4(v_texcoord, 0, 1);
```

Then you should get the following image:



We are assigning the U and V values from the texture coordinate data Red and Green color channels as shown here. Blue should be 0.
This way we can also see the problems with the pyramid and quad texture coordinates. We fix that in 3.

If your image with the above change does not look like this eg. everything is completely dark, then you most likely either do not assign anything to `v_texcoord` in the vertex shader or something is wrong in `setVertexAttributeArrays(...)`



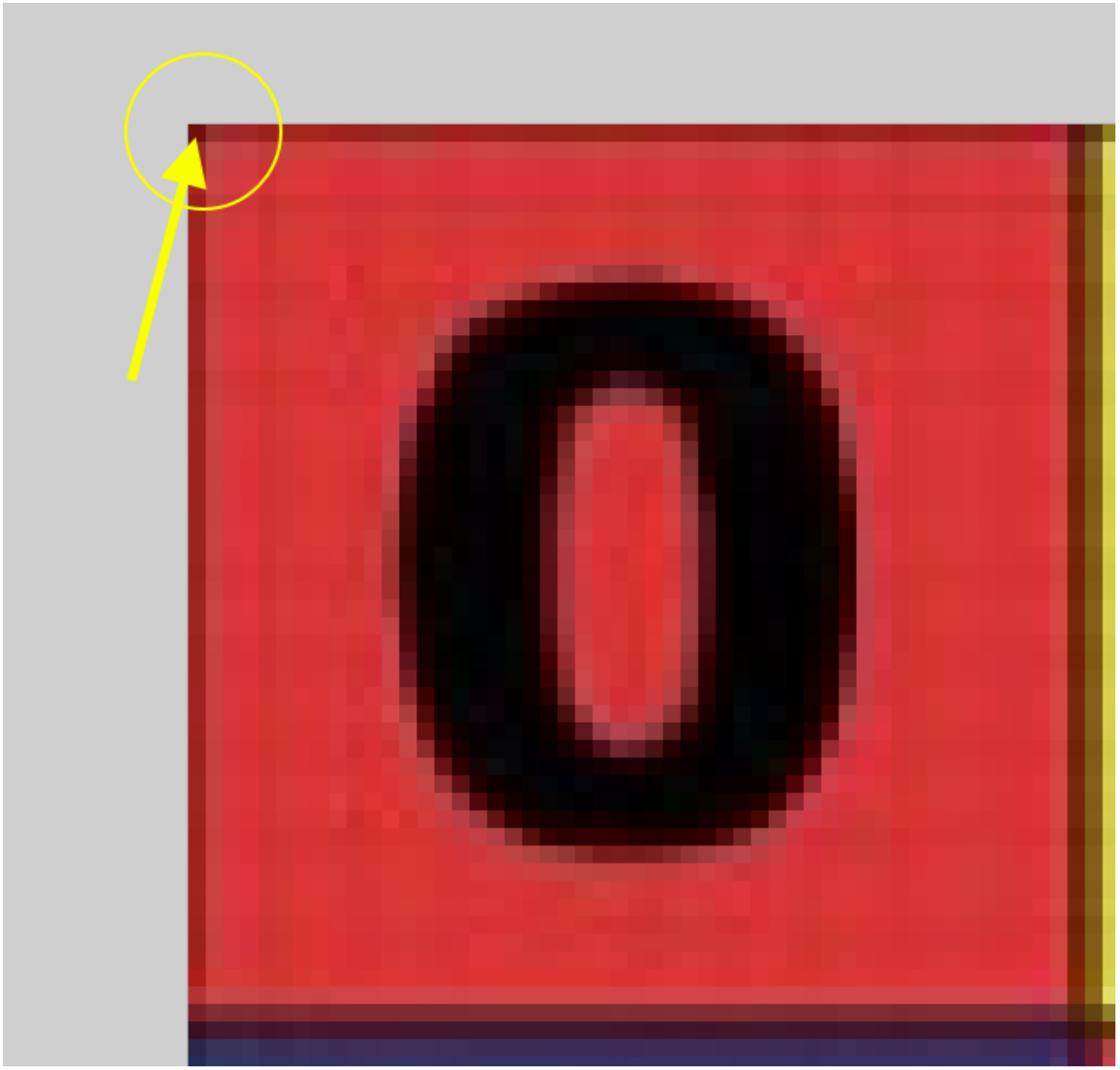
2.3 Assign a fixed texture coordinate

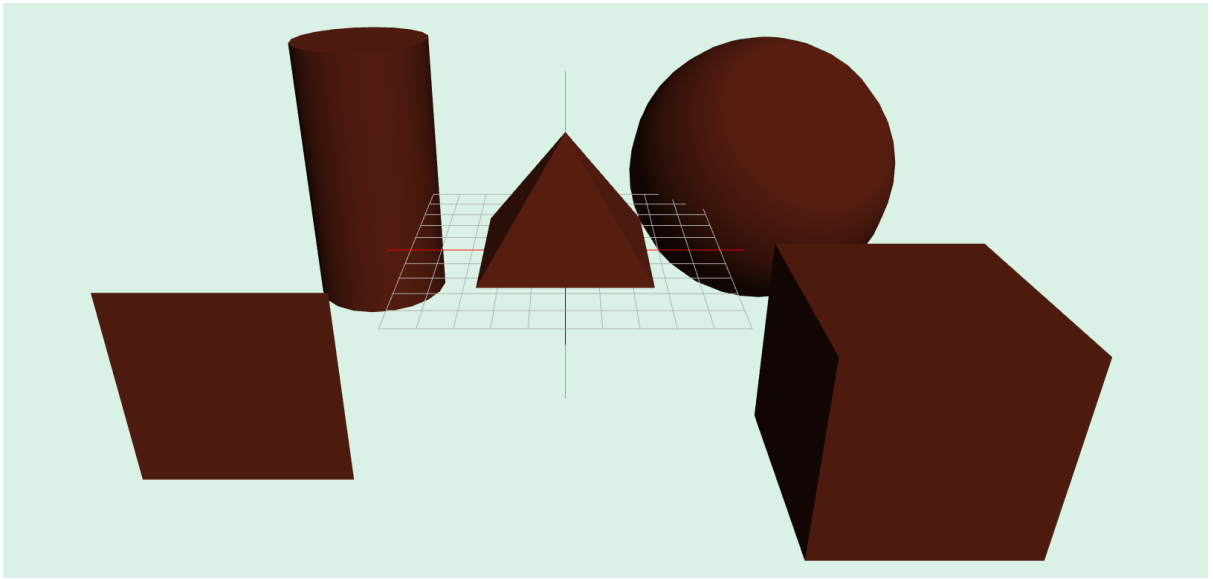
Instead of using `v_texcoord` you can also just assign a fixed coordinate for all pixels rendered by the fragment shader eg. [0.0, 1.0], when sampling the texture in the fragment shader:

```
vec3 textureColor = texture2D(u_mainTex, vec2(0.0,1.0)).rgb;
```

It will tell you if your texture is sampled at all, but `v_texcoord` is faulty.

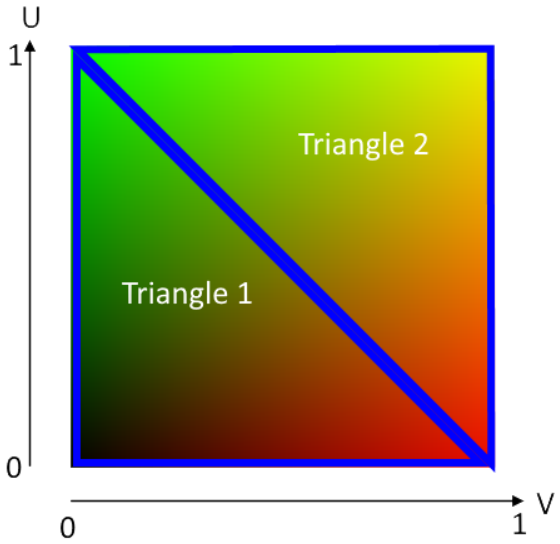
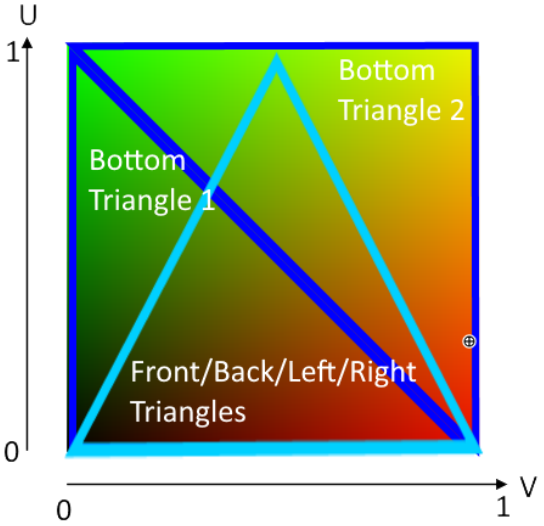
This will pick the topmost-leftmost pixel of the image:



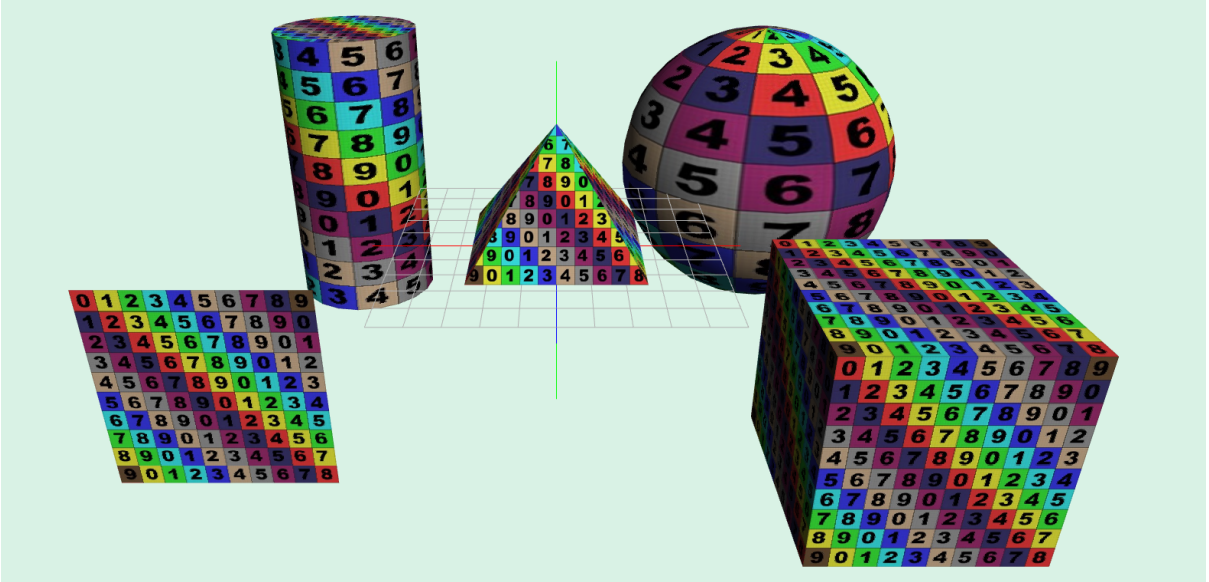


3. Fixing the Pyramid and Quad (30%)

Armed with the tools from 2., you are now able to fix the texture coordinate layout for the pyramid and quad primitive. Go to `primitives.js` and find the `Pyramid` and `Quad` there. The arrays for `texcoords` contain faulty data. Enter the correct data corresponding to these texture coordinate layouts:

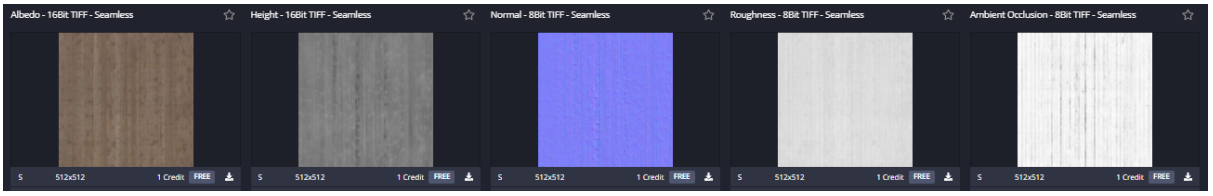
Quad UV Layout	Pyramid UV Layout
	

The completed scene with fixed texture coordinates then should look like this:



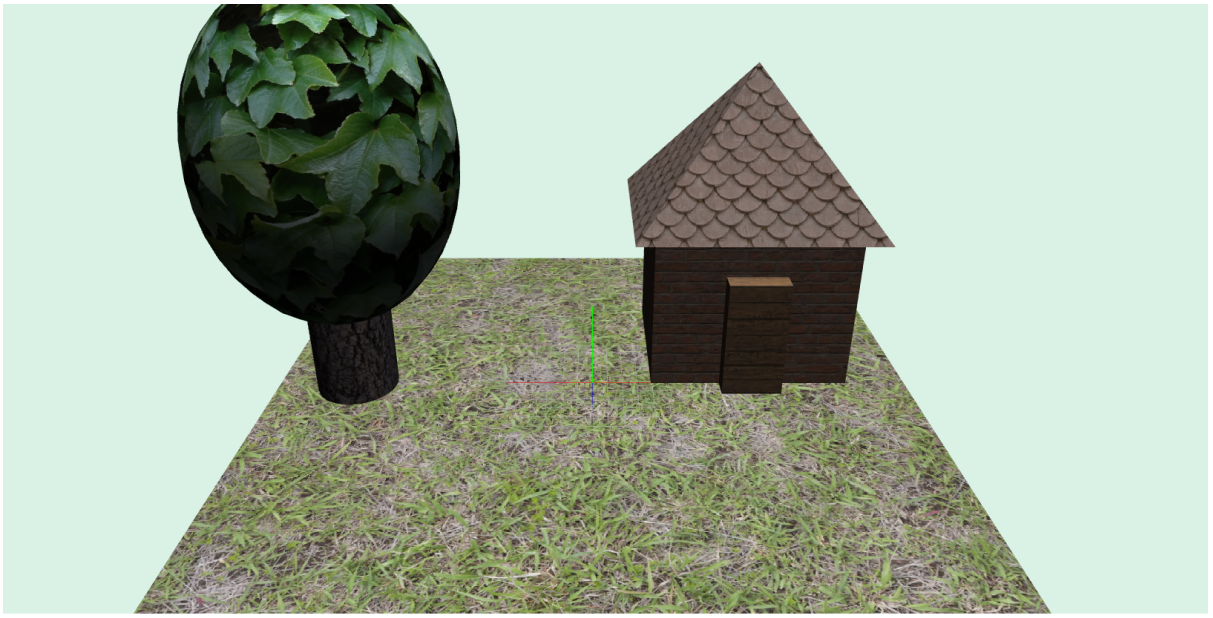
4. Setting up a Scene (10% + up to 10% bonus)

Let us now create something a bit more interesting and set up a few objects with nice textures.
You can find free textures at Textures.com: https://www.textures.com/ here you can download up to 12 free textures every day after registering for free.



*You can find setups with a lot of textures on Textures.com. We are only interested in the **ALBEDO or COLOR** texture. The other textures are used to create more realistic materials by influencing the way light interacts with the surface or how reflective, smooth or shiny a surface is in different spots. When using advanced shaders with normal-mapping, ambient occlusion, etc… these textures would be bound to Texture Units 1, 2, 3,….*

- Download several textures from textures.com.
    - Texture-size has to be a **power of 2 (eg. 128,256,512…) and square**.
    - The textures should be **no bigger than 1024x1024**. (submission size limit)
    - Only download the ALBEDO or COLOR textures.
- Load the textures in `init()` and create different materials.
- Create a scene with several objects using those materials.
    - There has to be at least a quad and a pyramid, to show the fixed UVs.
    - You may use the scene from the lighting lab, just copy the setup.
- Bonus points may be earned with outstanding scenes. Complete exercise 5:OBJ loader first.
- If you use a lot of individual objects you might run into performance issues, this is due to the debugger scripts: webgl-debug and webgl-lint. They take a severe toll on performance. To increase your framerate, disable them in engine.html. You won't get any more detailed error messages, but your program will run up to 100 times faster.



5. OBJ Model Loader (10%)

To complete this exercise, the last task is to integrate the model you created in the Blender Workshop into your scene. A common format to store 3D models is called OBJ developed by Wavefront Technologies. It is one of the simplest and most straightforward compressionless formats to store 3D data. A loader to create a mesh object from an .obj file is already included in your project.

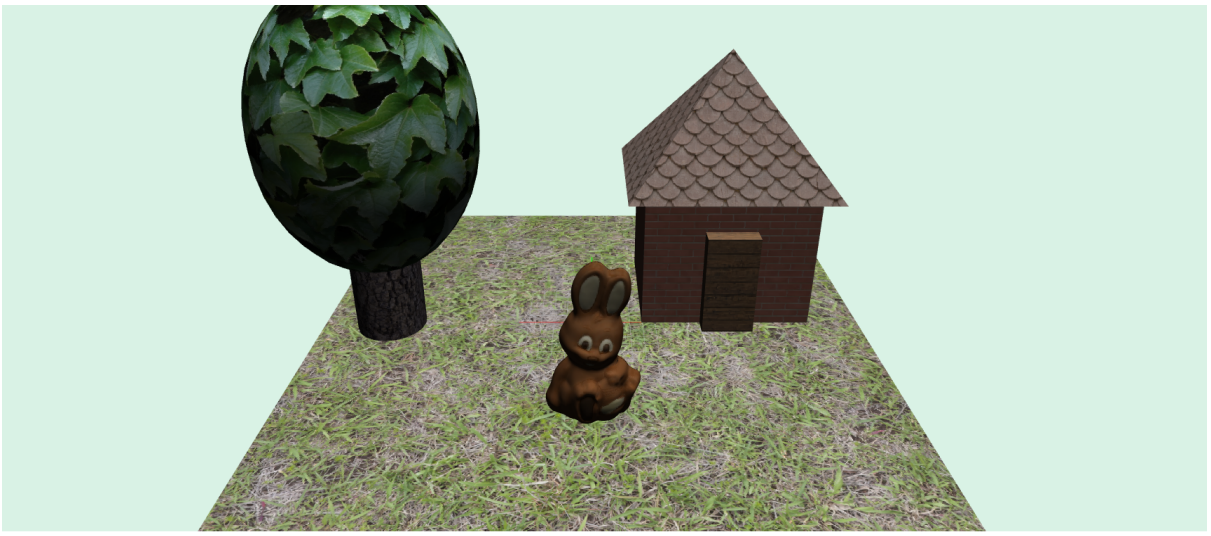
- Export your 3D model from Blender as an OBJ file.
    - Find instructions for this at Canvas/Files/Common Readings/ **Export OBJ Files In Blender.pdf**
- Include the OBJ Loader (parser) in your engine.html

```
<script src="../common/objparser.js"></script>
```

    - This will allow you to use the `OBJLoader` object to parse obj files and load the contained information into a mesh object.
- Load the OBJ file using the `OBJLoader`:

```
let bunnyMesh = OBJLoader.getMesh("bunnyMesh", "ressources/bunny.obj");
bunny = new ModelTransform(bunnyMesh, bunnyMat);
```

  Here with the example of a chocolate bunny obj file. The function `OBJLoader.getMesh(meshName, pathToMeshFile)` works the same as e.g. `Primitives.Cube.getMesh()`. It creates a mesh object with `meshName` in the `MeshCache`.
- Load the corresponding texture file you created in blender.
- Include the model in your scene



Should you **not have a model from Blender** or are unable to create one, you can download and **use the example bunny or pumpkin** from Canvas/Files/09-texturing and use it.