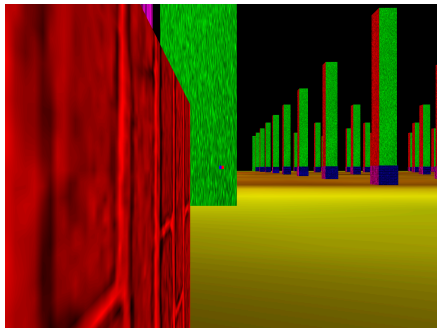
UNIVERSITÉ Concordia UNIVERSITY

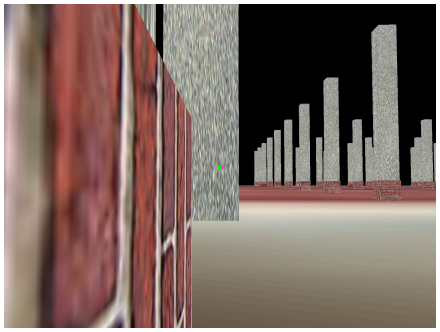
COMP 371
Computer Graphics

Lab 04 - Texture Mapping

Prepared by Nicolas Bergeron

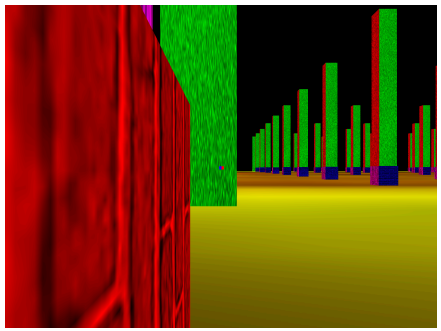
Expected results





Getting Started

- Download Lab01.zip from Moodle
- Download Lab04.zip and add lab04.cpp to the project (Visual Studio or Xcode)
- Add the Assets folder to the project as well
- After compiling and running the application, you should see the solution for lab03 + textures.



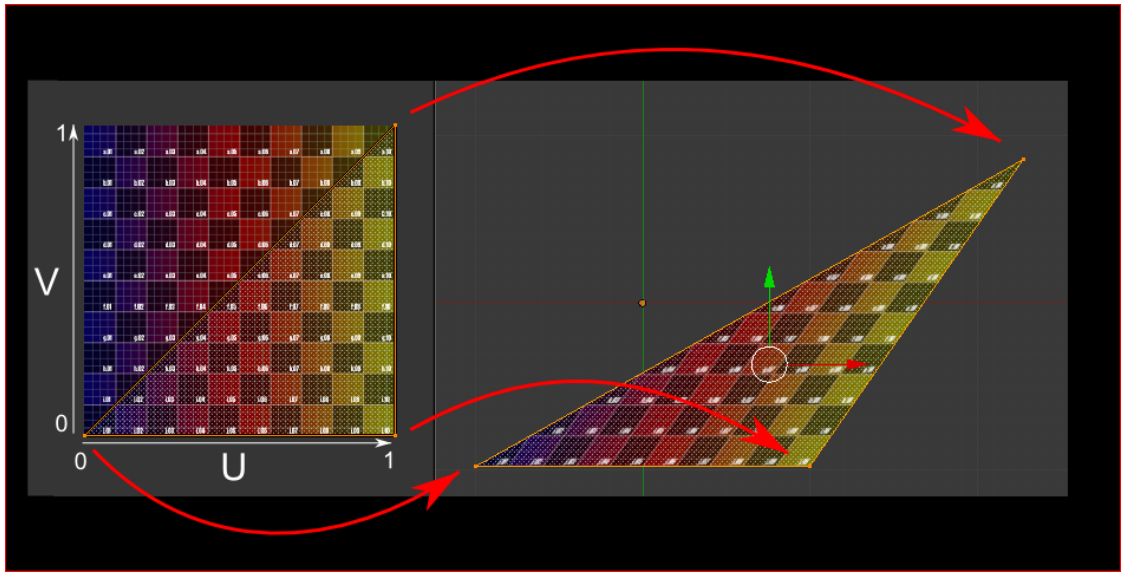
Outline for Lab04
Texture Mapping in OpenGL

- Load Images and upload them to the GPU
- Create a Vertex Buffer Object with UV coordinates
- Implement shaders supporting textures
- Draw Pillars base with brick texture, and top with cement texture
- Draw the avatar and projectiles without Texturing

TUTORIAL
TEXTURE MAPPING

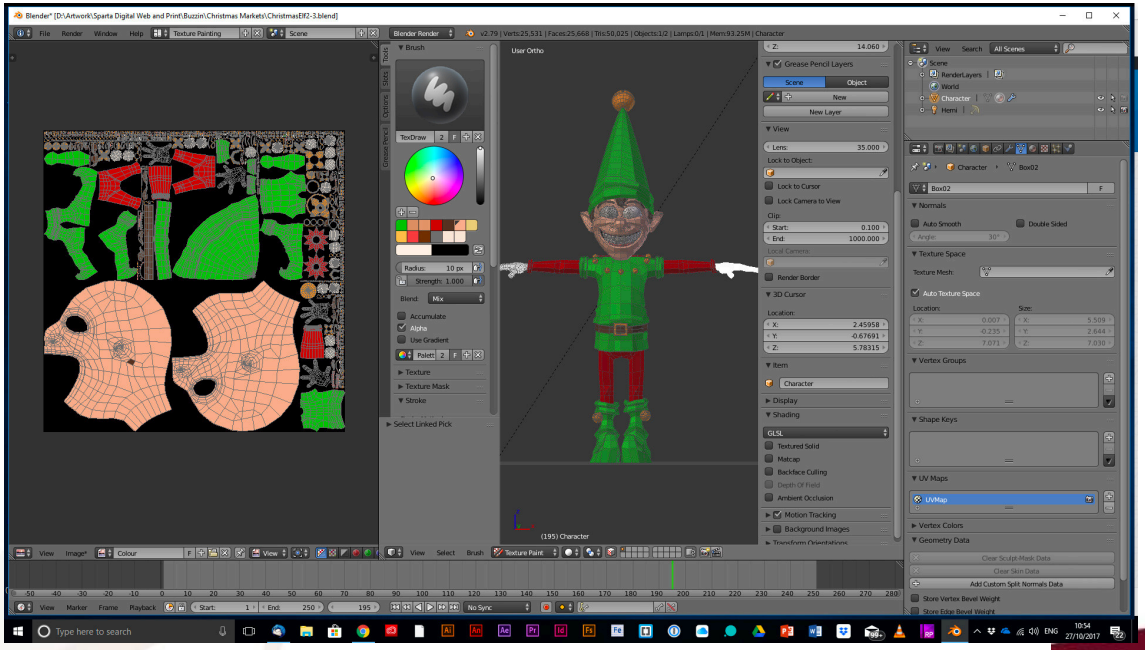
Texture Mapping with OpenGL

- A texture is an image in Computer Graphics. Texture mapping means applying an image on top of geometry.
- UV Coordinates represents which coordinates on an image will correspond with a vertex. Each vertex will contain a U coordinate and a V coordinate.



3D Models with Textures

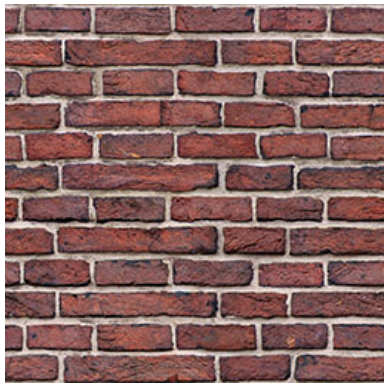
- In 3D models, a texture (left) contains multiple elements
- A 3D models (right) is bound to a single texture, the mapping of UV coordinates for each vertex is done by an artist.



Importing the texture

- Image formats support different features
  - BMP is simple to read, but high storage, no transparency
  - JPG is compact (good compression), no transparency
  - PNG supports transparency and compression
- In the framework, we use FreeImage, which support reading
- OpenGL likes power of 2 x power of 2 textures, such as 64x64, 256x256, 1024x1024, etc

Reading Textures with FreeImage and uploading them to GPU (VRAM)

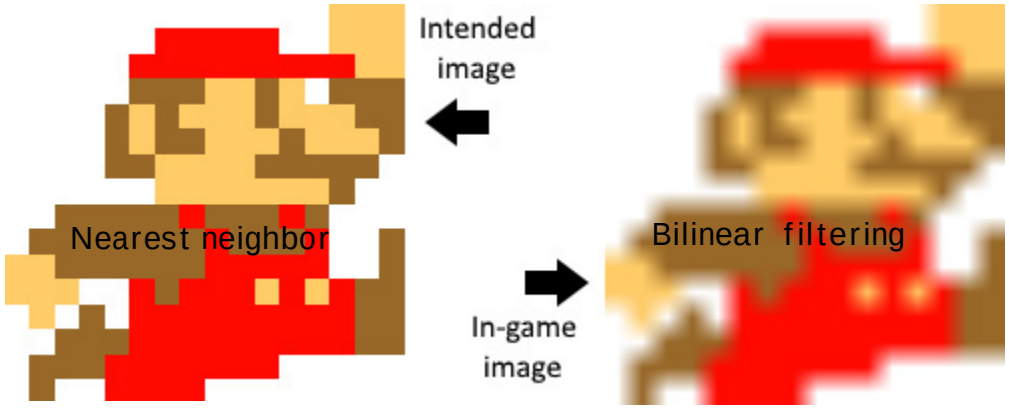


```
int loadTexture(char* imagepath)
{
    // Load image using the Free Image library
    FREE_IMAGE_FORMAT format = FreeImage_GetFileType(imagepath, 0);
    FIBITMAP* image = FreeImage_Load(format, imagepath);
    FIBITMAP* image32bits = FreeImage_ConvertTo32Bits(image);

    // Get an available texture index from OpenGL
    GLuint texture = 0;
    glGenTextures(1, &texture);
    assert(texture != 0);

    // Set OpenGL filtering properties (bi-linear interpolation)
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    // Retrieve width and hight
    int width = FreeImage_GetWidth(image32bits);
    int height = FreeImage_GetHeight(image32bits);

    // This will upload the texture to the GPU memory
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height,
                 0, GL_BGRA, GL_UNSIGNED_BYTE, (void*)FreeImage_GetBits(image32bits));

    // Free images
    FreeImage_Unload(image);
    FreeImage_Unload(image32bits);

    return texture;
}
```

libFreeImage decodes the image file and converts it to RGBA 32-bits

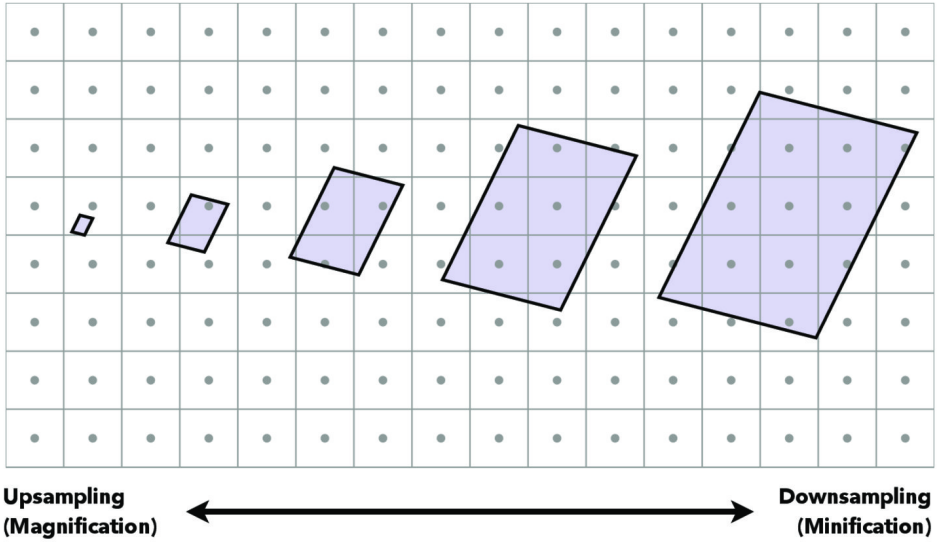
OpenGL generates a unique texture id, sets texture filtering options and uploads the image to VRAM

Filtering Algorithms



↑Source: https://devforum.roblox.com/t/option-to-turn-off-bilinear-filter-on-images/29035

Screen Pixel Footprint in Texture



←Source: https://cs184.eecs.berkeley.edu/

Adding UV coordinates to Vertex Data

- We must include UV coordinates with vertex data
- It is convenient to use a class or struct to layout the data
- In more complex examples, different models use different vertex data

```
struct TexturedColoredVertex
{
    TexturedColoredVertex(vec3 _position, vec3 _color, vec2 _uv)
    : position(_position), color(_color), uv(_uv) {}

    vec3 position;
    vec3 color;
    vec2 uv;
};

// Textured Cube model
const TexturedColoredVertex texturedCubeVertexArray[] = {  // position,
    TexturedColoredVertex(vec3(-0.5f,-0.5f,-0.5f), vec3(1.0f, 0.0f, 0.0f), vec2(0.0f, 0.0f)),
    TexturedColoredVertex(vec3(-0.5f,-0.5f, 0.5f), vec3(1.0f, 0.0f, 0.0f), vec2(0.0f, 1.0f)),
    TexturedColoredVertex(vec3(-0.5f, 0.5f, 0.5f), vec3(1.0f, 0.0f, 0.0f), vec2(1.0f, 1.0f)),

    TexturedColoredVertex(vec3(-0.5f,-0.5f,-0.5f), vec3(1.0f, 0.0f, 0.0f), vec2(0.0f, 0.0f)),
    TexturedColoredVertex(vec3(-0.5f, 0.5f, 0.5f), vec3(1.0f, 0.0f, 0.0f), vec2(1.0f, 1.0f)),
    TexturedColoredVertex(vec3(-0.5f, 0.5f,-0.5f), vec3(1.0f, 0.0f, 0.0f), vec2(1.0f, 0.0f)),

    TexturedColoredVertex(vec3( 0.5f, 0.5f,-0.5f), vec3(0.0f, 0.0f, 1.0f), vec2(1.0f, 1.0f)),
    TexturedColoredVertex(vec3(-0.5f,-0.5f,-0.5f), vec3(0.0f, 0.0f, 1.0f), vec2(0.0f, 0.0f)),
    TexturedColoredVertex(vec3(-0.5f, 0.5f,-0.5f), vec3(0.0f, 0.0f, 1.0f), vec2(0.0f, 1.0f)),
```

VBO changes

- We add the UV data to our VBO, to specify where the UV coordinates are:

```
glVertexAttribPointer(2,                              // attribute 2 matches aUV in Vertex Shader
                      2,
                      GL_FLOAT,
                      GL_FALSE,
                      sizeof(TexturedColoredVertex),
                      (void*)(2*sizeof(vec3))      // uv is offseted by 2 vec3 (comes after position and color)
                      );
glEnableVertexAttribArray(2);
```

Shaders for textured Geometry

```
const char* getTexturedVertexShaderSource()
{
    // For now, you use a string for your shader code, in the assignment, shaders will b
    return
    "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;"
    "layout (location = 1) in vec3 aColor;"
    "layout (location = 2) in vec2 aUV;"
    ""
    "uniform mat4 worldMatrix;"
    "uniform mat4 viewMatrix = mat4(1.0);"  // default value for view matrix (identity)
    "uniform mat4 projectionMatrix = mat4(1.0);"
    ""
    "out vec3 vertexColor;"
    "out vec2 vertexUV;"
    ""
    "void main()"
    "{"
    "   vertexColor = aColor;"
    "   mat4 modelViewProjection = projectionMatrix * viewMatrix * worldMatrix;"
    "   gl_Position = modelViewProjection * vec4(aPos.x, aPos.y, aPos.z, 1.0);"
    "   vertexUV = aUV;"
    "}";
}
```

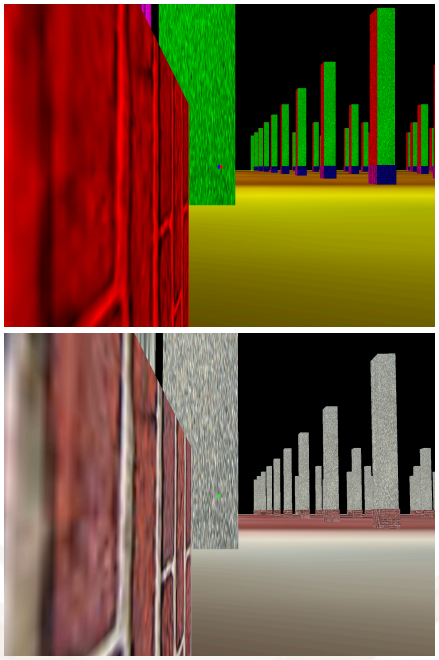
Notice the vertex shader receives UV attributes and outputs them back to fragment shader

Fragment shader fetches the texture sample at coordinate UV and blends the color with vertex colors.

```
const char* getTexturedFragmentShaderSource()
{
    return
    "#version 330 core\n"
    "in vec3 vertexColor;"
    "in vec2 vertexUV;"
    "uniform sampler2D textureSampler;"
    ""
    "out vec4 FragColor;"
    "void main()"
    "{"
    "   vec4 textureColor = texture( textureSampler, vertexUV );"
    "   FragColor = textureColor * vec4(vertexColor.r, vertexColor.g, vertexColor.b, 1.0f);"
    "}";
}
```

Keep only texture color in fragment shader

- Blending vertex colors and texture coordinates is useful for some effects, but we can also just use the texture color for this scenario…



```
const char* getTexturedFragmentShaderSource()
{
    return
    "#version 330 core\n"
    "in vec3 vertexColor;"
    "in vec2 vertexUV;"
    "uniform sampler2D textureSampler;"
    ""
    "out vec4 FragColor;"
    "void main()"
    "{"
    "    vec4 textureColor = texture( textureSampler, vertexUV );"
    "    FragColor = textureColor;"
    "}";
}
```

Drawing the same model with different textures

- We can bind different textures to use with the same VBO and shaders. In our case, the base of a pillar is using the brick texture, while the top uses the cement texture.

```
for (int i=0; i<20; ++i)
{
    for (int j=0; j<20; ++j)
    {
        // FIXME: it would be more efficient to set the cement texture and draw all pillars, then switch to brick texture and
        // use cement texture for pillar
        glBindTexture(GL_TEXTURE_2D, cementTextureID);
        pillarWorldMatrix = translate(mat4(1.0f), vec3(- 100.0f + i * 10.0f, 5.0f, -100.0f + j * 10.0f)) * scale(mat4(1.0f), v
        setWorldMatrix(texturedShaderProgram, pillarWorldMatrix);
        glDrawArrays(GL_TRIANGLES, 0, 36);

        // use brick texture for base
        glBindTexture(GL_TEXTURE_2D, brickTextureID);
        pillarWorldMatrix = translate(mat4(1.0f), vec3(- 100.0f + i * 10.0f, 0.55f, -100.0f + j * 10.0f)) * rotate(mat4(1.0f),
            scale(mat4(1.0f), vec3(1.1f, 1.1f, 1.1f));
        setWorldMatrix(texturedShaderProgram, pillarWorldMatrix);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
}
```

Draw model with different shaders (with and without textures)

- The avatar (model shooting projectiles) and projectiles can be drawn by using the color shader. This will ignore the texture coordinates and fragment shader returns the color of the vertices.

```
// Draw colored geometry
glUseProgram(colorShaderProgram);

// Update and draw projectiles
for (list<Projectile>::iterator it = projectileList.begin(); it != projectileList.end(); ++it)
{
    it->Update(dt);
    it->Draw();
}
```

Exercise

- To learn more about Texture Mapping, it is recommended to go through these tutorials and understand the concepts more in depth
- https://learnopengl.com/Getting-started/Textures
- http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube