CAP 5705 – Final Project

Report

Nathan Harris (5438-7588) Abhinav Aryal (3327-1507)

1. Scene

For the final project, we used three different object files to observe every type of observation during the process. We used a cube to start our process and finally used objects with many more vertices. We used three different obj files for the final demonstration.
In the pictures below you can observe various obj files with three different texture files, the first texture file is the brickwall that can be seen on the cube, the send is the wooden texture that can be seen on the Titanic, and finally, the gator can be seen with the alligator object.

Figure: Cube with brick wall texture

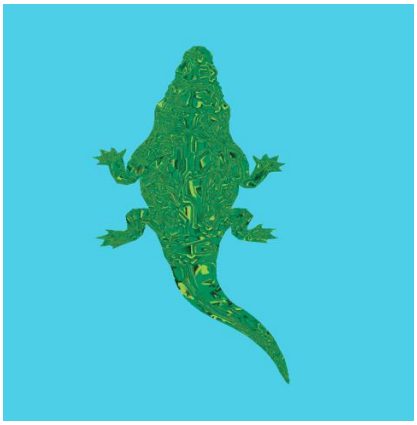

Figure: Gator with scale texture

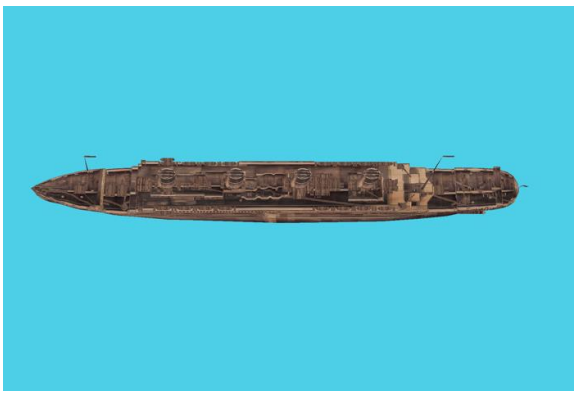

Figure: Titanic with wooden texture

2. Goals of the Project

For this project, we want to implement 2 different rendering features using textures. Texture mapping, where an image texture is projected onto a rendered model, and Normal mapping, where a textured surface is added to a model, affecting the lighting on the model. We also tried to implement a theme for UF by implementing various textures to the gator that we imported from the crocodile.obj file.

Timeline of project activities

- **Nov 16th - Nov 20th:** Research work on Project
- **Nov 20th - Nov 26th:** Finalize resources required for the project
- **Nov 27th:** Start project
- **Dec 8th:** Finish Texture mapping, and begin Normal mapping
- **Dec 15th:** Finish project work and complete the documentation.

Survey of existing literature/tools

- **"Texture and reflection in computer generated images":**
  Blinn provided an example of a satellite during his research. It was combined with an environment map that showed the sun and the planet. Notably, these items were exposed to conventional light in both cases, which added to their realistic appearance by creating diffuse lighting effects.[1]

- **"Environment Mapping and Other Applications of World Projections":**
  During this research, Greene combined a computer-generated image of the ground with a skyshot to create an environment map. The rendering on the right used a method that converted the fisheye picture back into a perspective view. This conversion accomplished two goals at once: it immediately produced a perspective representation of the surrounding area and projected it onto the ship in the scene.[2]

- **"Real-Time Rendering" [3]:**

Tomas Akenine-Möller, Eric Haines, and Naty Hoffman's book "Real-Time Rendering" is an essential tool for computer graphics. It covers bump mapping and normal mapping in brief, providing clear but thorough explanations of their theoretical foundations, useful implementation techniques, and significant contributions to the improvement of produced scenes' realism in real-time graphics applications.[3]

- **"Normal Mapping":**

  The idea of normal mapping, a method for improving 3D surface realism by mimicking fine features on flat objects, is presented in this lesson along with its application in OpenGL. It covers the basics of using shaders to guarantee accurate lighting calculations, converting normal vectors into tangent space, and modifying per-fragment normals using textures. It also covers how to apply normal mapping to complicated models, calculate tangent and bitangent vectors manually, and reduce the number of vertices in order to optimise efficiency. Using the Gram-Schmidt approach, the text addresses quality improvements in its conclusion. All in all, it provides a thorough grasp of the theory, practice, and optimisation of normal mapping in 3D rendering.[4]

- **GLSL Shaders:**
  An essential component of typical mapping is GLSL (OpenGL Shading Language). With it, programmers may design unique shaders to work with normal maps, convert normal vectors into tangent space, and compute illumination using the adjusted normal.

- **Texture Mapping Functions:**
  OpenGL has texture mapping functions essential for normal mapping 3D objects. Per-fragment normal adjustments are made possible by these routines, which allow normal map textures to be loaded and sampled onto surfaces in shaders.

- **OpenGL Mathematics (GLM)**:

It is a mathematics library written in C++ that aims to emulate the capabilities of the math functions in GLSL. It's beneficial for handling vector operations needed for regular mapping computations, calculating tangent and bitangent vectors, and executing matrix transformations.

- **GLFW, GLEW and OpenGL FrameWorks:**
  These frameworks offer tools to manage OpenGL contexts, handle window generation, and receive user input. They may not be directly connected to normal mapping, but they serve as the foundation for developing OpenGL applications that make use of normal mapping techniques.

3. Implementation

3.1. Texture Mapping

For the first part of our final project, we implemented texture mapping to our renderer. The core idea behind texture mapping is that, when the fragment shader is deciding what colour to use for each fragment, the shader pulls the surface colour of the object or model from a 2D image (texture) that has been pre-loaded. Several parts need to be put in place to execute this idea.

Firstly, the texture needs to be loaded into the program. STB, an open-source library, makes this easy. STB contains functions to load in several image formats so that OpenGL can interface with them. Starting at line 155 in 'base. cpp' we load in our textures using STB and then write them to a uniform buffer so that they may be accessed by the shaders in the GPU.

Secondly, the object file needs to define how a texture is to be mapped to each of its vertices. We added to our 'cube.obj' file several 'vt' tags, one for each vertex of the cube, that specified which coordinate of the texture should be mapped to that vertex. Here are the mappings we used

```
vt 1 1  //front bottom left corner
vt 1 0  //front upper left corner
vt 0 0  //front upper right corner
vt 0 1  //front bottom right corner
vt 0 0  //back bottom left corner
vt 0 1  //back upper left corner
vt 1 1  //back upper right corner
vt 1 0  //back bottom right corner
```

*There is one shortcoming of using these mappings, one that I will cover later in this section.

Then, after adding the new vt tags to our cube model, we needed to update our 'importObj()' and shader buffers to account for the added vertex texture data. If the imported model contained vt tags, those coordinates would be appropriately added to the output array of data. Additionally, we needed to create a buffer to hold that data, separate from the buffers already holding the vertex position data, color data, etc.

Lastly, we needed to use the data in the texture and vertex buffers inside the fragment shader to correctly color the fragment. This is as easy as creating uniform and input variables in the fragment and vertex shader so that, in the fragment shader, we can run the following line of code to set the fragment color correctly:

```
//texture1 is our loaded image texture
//TexCoord is where on the image the current fragment should
pull its colour from
//FragColor is the output value, the final colour of the
fragment.
FragColor = texture(texture1, TexCoord);
```

3.2. Wrapping texture around the entire cube

One issue with implementing texture mapping we had was that, because our renderer was using indexed triangles to store the model data, where there is only one instance of a vertex for all the faces it is a part of, we could only assign one texture coordinate to each face. This is an issue because, in our effort to map copies of the same brick texture to each side of our cube, we would need to tell the same vertex to map to a different part of the texture for each face it is a part of. But we only have 1 instance of the vertex in the buffer. Therein lies the limitation, which to fix we would have had to revert our wavefront file parser and renderer to support the separated triangles method of storing model data.

```
vt 0 0  //front bottom left corner
vt 0 1  //front upper left corner
vt 1 1  //front upper right corner
vt 1 0  //front bottom right corner
vt 0 0  //back bottom left corner
vt 0 1  //back upper left corner
vt 1 1  //back upper right corner
vt 1 0  //back bottom right corner
```

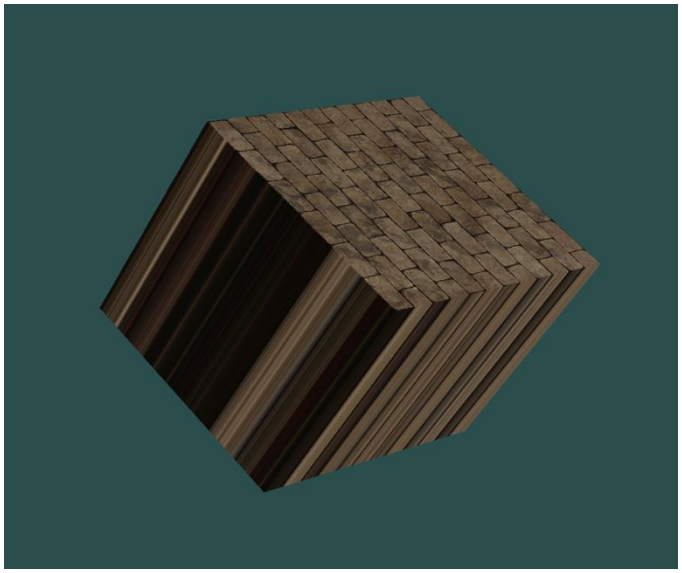

Figure: Cube before making the texture adjustment

One workaround we found was that we could stretch the existing texture around corners to cover all sides of the cube, as shown below

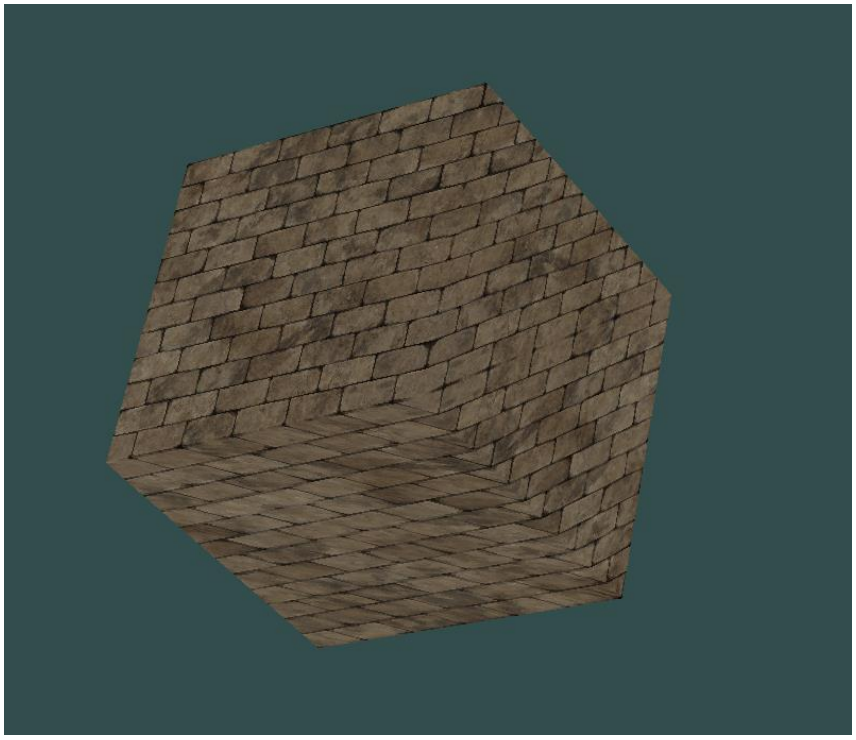

Figure: Cube after making the texture adjustment

There are 2 shortcomings with this workaround.

1) the texture is stretched and sheared in a visually obvious way, and
2) the lighting for the texture does not match the face it is projected to, so lighting is not accurate when the model is rotated.

3.3. Normal mapping

In the second part of our project, we implemented normal mapping. As normal maps are still textures, importing the normal map textures into the renderer and shaders is identical to the texture mapping part, so I will gloss over that for now. Inside the fragment shader, we combined our existing flat shading code with the normals of the normal map to achieve the textured brick wall appearance.

Our normal map assumes the default vector is (0,0,1), or a unit vector in the z direction. However, the face that the normal map is projected onto will not always be facing the z direction, so we need to create a rotation matrix to properly account for any rotation applied to the model. We create this matrix inside 'shaders/texturenormal.fs' with

```
vec3 xTangent = dFdx(viewPos);
vec3 yTangent = dFdy(viewPos);
vec3 faceNormal = normalize(cross(xTangent, yTangent));
vec3 T = normalize(vec3(model * vec4(xTangent, 0.0)));
vec3 B = normalize(vec3(model * vec4(yTangent, 0.0)));
vec3 N = normalize(vec3(model * vec4(faceNormal, 0.0)));
mat3 TBN = mat3(T,B,N);
```

where T,B, and N stand for tangent, bitangent, and normal vectors.

We pull our normal from the normal map like so, scaling and offsetting the texture value appropriately:

```
vec3 normal = texture(texture2, TexCoord).rgb *2.0f - 1.0f;
```

Then multiply the TBN matrix by the normal vector to get the normal, correctly rotated to match the rotation of the face of the model.

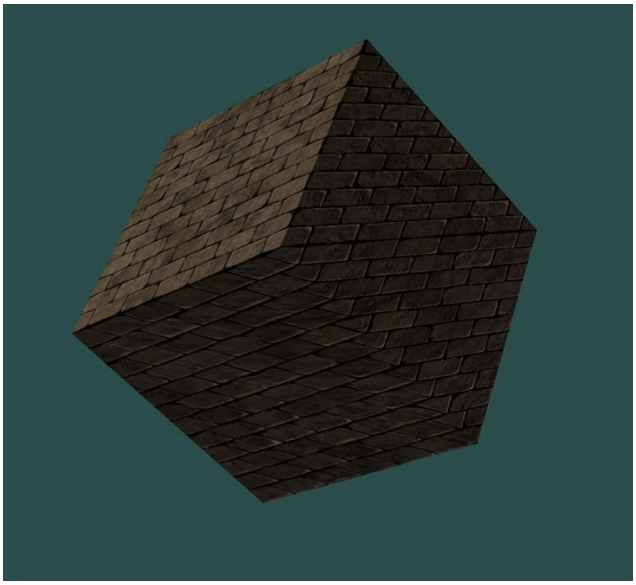

Figure: Normal mapping on the cube.obj

3.4. The issue with normal mapping before using TBN matrix:

We had implemented the normal mapping before we had used TBN matrix but this led to us only achieving a little difference and didn't the change that we wanted to after any kind of transformation. This issue was solved by the above-mentioned method by using the TBN matrix.

3.5. Switching through various modes

Additionally inside 'texturenormal.fs' we allow the user to toggle the brick normal map off or on, and choose between different textures to see how the mapping processes change with different assets.

As can be seen in the scene section, we have multiple textures and 2 types of mapping as mentioned above,

We have made the program user-friendly to allow the user to change the texture and allow the user to turn on and off the normal mapping.

To show the implementation, we are using the titatnic. obj and showing the difference, this has also been shown in the demonstration video.

Here we are running the program by:

./a.out titanic.obj texturenormal

Default (or Pressing j from any other mode)

Figure: Titanic.obj under normal texture

Pressing M(turning off the normal mapping)

Figure: Titanic.obj without normal texture mapping

Pressing K/L to change the texture

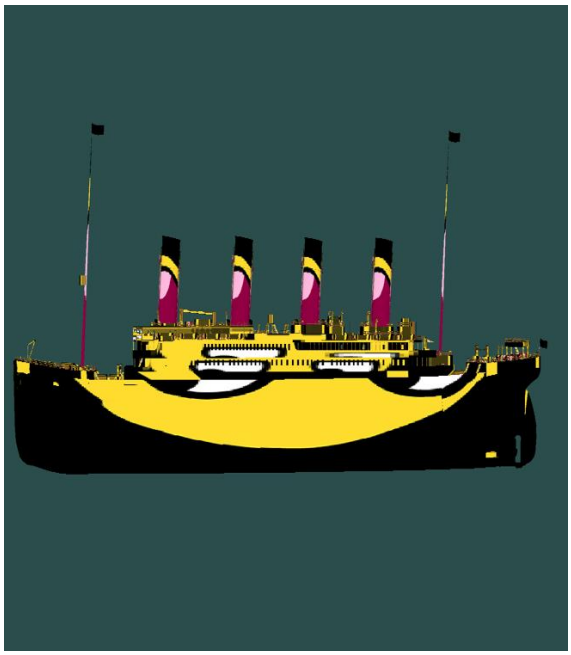

Figure: With Texture smile

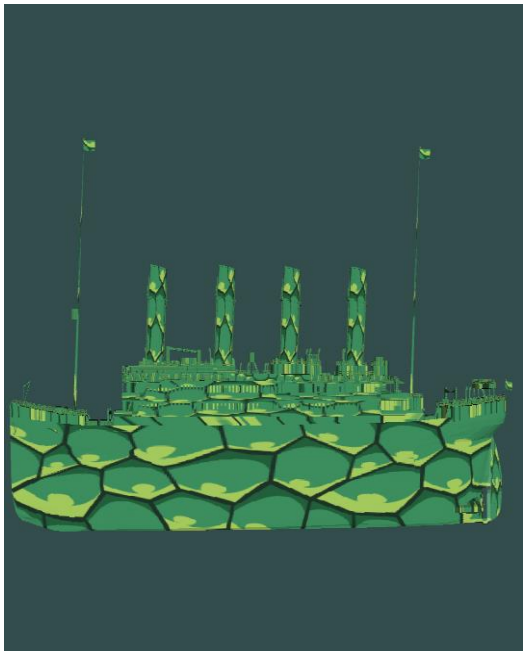

Figure: With Texture of gator skin

4. Learning outcome:

The effective use of texture mapping and normal mapping techniques is the result of extensive study and hands-on experience with sophisticated rendering techniques. Using the STB library to load textures, handling vertex texture coordinates, and sending texture data to shaders quickly and effectively are examples of controlling texture-related components of the rendering process. Overcoming difficulties brought about by the allocation of texture coordinates' indexed triangle restriction further demonstrates flexibility and problem-solving skills in handling complexities in the rendering process.

Additionally, the addition of normal mapping—which makes use of the TBN (Tangent, Bitangent, Normal) matrix—indicates a more thorough comprehension of surface detail simulation through the manipulation of normal vectors using texture data. Overcoming obstacles in the process of changing normal vectors between various spaces indicates a solid understanding of mathematical ideas, especially those connected to geometric transformations and linear algebra. This issue's effective resolution highlights the need

to combine technical know-how with analytical reasoning when handling challenging problems.

The thorough application of both texture mapping and normal mapping techniques highlights expertise in shader creation and shows an understanding of optimisation factors and performance improvements that are essential in contemporary graphics programming. In addition, recording problems encountered and solutions developed demonstrates proficient communication abilities and the capacity to distribute information, promoting cooperative learning settings.

This accomplishment demonstrates a comprehensive grasp of sophisticated rendering methods, including texture and normal mapping, shader integration, geometric transformations, problem-solving techniques, and effective communication. It also highlights a broad knowledge of computer graphics programming.

With the effective integration of texture mapping and normal mapping into the renderer, this enhanced learning result highlights the complex character of your project and elaborates on a variety of technical features, problem-solving abilities, and a thorough grasp of rendering methodologies.

Along with these two mapping techniques we also learned about environmental mapping in depth but its implementation was something that we weren't able to complete due to time constrain, we gained valuable knowledge about it and environmental mapping is something that we will we working on, more on this can be seen in the future work section.

5. Work distribution

For the overall implementation of the project we have imagined having various tasks that we will have to complete, we have distributed the work in such a way that both of us will be able to learn most of the things we will be able to do during the project.
At the very start, both of us did our own research to be sure that we got to learn about various topics before we started with the project.

As we finalize our project to be hovering around texture mapping, we have decided to distribute the work in the following manner:

For the work we will be doing during texture mapping:

Nathan:

- Created colour textures of varying resolutions
- Programmed to read in texture file in importObj()

Abhinav:

- Programmed to Pull pixel data from loaded tex file using vt coords
  `texture.getPixelColor(u,v);`
- Replaced data in color buffer with colors loaded from texture
- Rendered texture map onto the model!!!!

```
FragColor = texture(texture1, TexCoord);
```

And after that for Normal mapping:

Nathan:

- Created normal map textures
- Added buffer to store normals
- Pulled and stored normal map values from normal map texture in importObj()
- Switching between multiple textures by pressing various keys

```
if (texCycle == 0) {
    FragColor = vec4(lightCoef * texture(texture1, TexCoord).rgb, 1.0);
}
else if (texCycle == 1) {
    FragColor = vec4(lightCoef * texture(texture3, TexCoord).rgb, 1.0);
}
else if (texCycle == 2) {
    FragColor = vec4(lightCoef * texture(texture4, TexCoord).rgb, 1.0);
}
```

Abhinav:

- Added normal buffer to shaders
- Use per-fragment normal to adjust lighting
  `float lightCoef = dot(lightDir, fragNormal);`

```
vec3 xTangent = dFdx(viewPos);
vec3 yTangent = dFdy(viewPos);
vec3 faceNormal = normalize(cross(xTangent, yTangent));
vec3 T = normalize(vec3(model * vec4(xTangent, 0.0)));
vec3 B = normalize(vec3(model * vec4(yTangent, 0.0)));
vec3 N = normalize(vec3(model * vec4(faceNormal, 0.0)));

mat3 TBN = mat3(T,B,N);

vec3 normal = texture(texture2, TexCoord).rgb *2.0f - 1.0f;
normal = normalize(TBN *normal);
```

Work for Documentation and Report Writing was done by both, Abhinav handled the README file and made the video demonstration while Nathan wrote about the implementation.

6. Future Work

In future, as it progresses, the project provides several opportunities for improvement and more refinement. Future research may look into how to include environmental mapping methods in the renderer. Environmental mapping includes a range of techniques, including spherical and cube mapping, that mimic the surrounding environment's reflections and refractions.

By adding realistic reflections and ambient lighting, environmental mapping would greatly improve the generated scenes' visual reality. This might entail gathering information about the skybox, surrounding area, and other objects, and then utilising that data to generate reflections on the scene's surfaces.

Furthermore, rendering capabilities may be significantly improved by extending the renderer to handle more varied materials and surface attributes using sophisticated shading models, including metallic or specular surfaces. It may be possible to simulate more realistic materials with correct interactions between light, surface roughness, and reflections by using physically-based rendering (PBR) approaches.

Additionally, there may be additional research[6] done on how to overcome the restriction about indexed triangles for the allocation of texture coordinates. This restriction may be addressed by implementing sophisticated mesh data structures or using different methods for storing texture coordinates for each face. This would enable more flexible and thorough texture mapping on objects that have common vertices.

7. UF Theme:

For the UF Theme we tried to map various textures on a mesh that creates a gator.

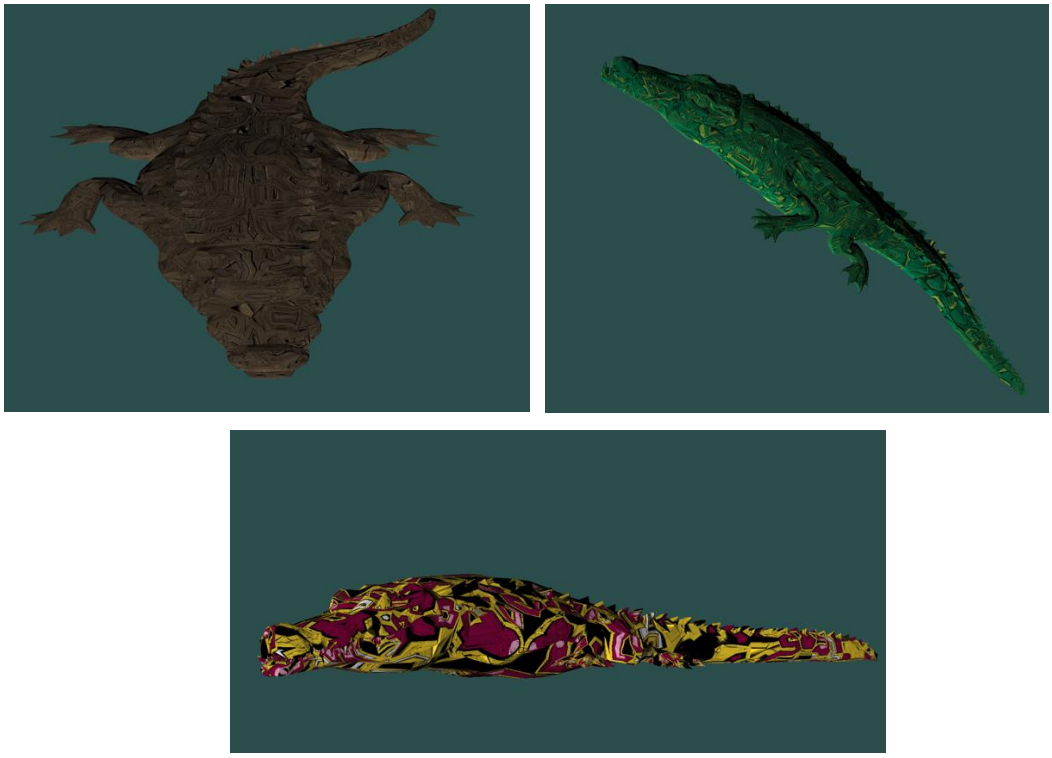

Figure: Gator with multiple textures

Reference

1. Blinn, J. F. and Newell, M. E. Texture and reflection in computer generated image Communications of the ACM Vol. 19, No. 10 (October 1976), 542-547

2. Ned Greene. Environment Mapping and Other Applications of World Projections. IEEE Computer Graphics and Applications, Vol 6. No. 11. Nov. 1986.
3. "Real-Time Rendering" by Tomas Akenine-Möller, Eric Haines, Naty Hoffman
4. https://learnopengl.com/Advanced-Lighting/Normal-Mapping
5. https://learnopengl.com/Advanced-Lighting/Textures
6. https://www.pauldebevec.com/ReflectionMapping/
7. A general method for preserving attribute values on simplified meshes. Cignoni: C. Montani, C. Rocchini, R. Scopigno