Crystallographic Sphere Renderer

Suvi, Carolyn, Phillip, Griffin

Introduction:

Our project started out with a simple idea. Create a realistic and beautiful looking crystallographic sphere. So how did we get this idea? First, we talked as a group about our common interests, and this included perlin noise, toon shading and bump maps.

And because we noticed that we liked shading, we came up with a shared vision: a sphere with a crystalline appearance and lighting. As the project progressed, we added more on adjusting the output with UI, and adding other objects.

Renderer Development

As part of bringing together different shading methods into one final product, a single rendering interface was required. We wanted our program to flexibly support displaying different shading methods working together and we wanted the ability to turn these shading modes on and off at will. We took Carolyn's original shading code and refactored it from one long Python script from into four different modules:

- main.py
- scene_setup.py
- file_handler.py
- shading_lib.py

Part of the reorganization involved cutting out redundant code, renaming variables for more clarity, and the refactoring of global variables into function parameters, which many parts of the original script were dependent on. Ultimately, the renderer was refactored with clarity and separation of concerns in mind. We opted to preserve the main rendering pipeline while abstracting away functions and redundant code into functions and other modules.

In order to turn shading features on and off, we needed to create separate flags for each shading mode (normal mapping, texture mapping, toon shading, shadow mapping; etc). These separate flags were then used as "handles" by our UI, which could then freely modify the shading parameters and return the updated output to the screen.

Renderer Architecture

Fig. 1: Architecture of our renderer, with the UI Framework "Streamlit" on the frontend

File_handler: a library of functions and classes developed to parse scene files, 3d object files, and texturing attributes into objects for the renderer (main.py) to use. File_handler abstracts away the code required to open files and simplifies the interaction needed from the user to open a file. The functions included automatically attempt to open an alternative backup URL in case the passed in object file cannot be found. The module provides written output and fails gracefully if all attempts to open the file fail.

Scene_setup: a library of functions relating to matrix setup and creation. Used as part of initializing the Camera Matrix, Perspective Projection Matrix and Transformation Matrices for the scene.

Shading_lib: a library containing all shading calculation functions performed by the renderer. The library also comes with helper functions and classes to assist with the calculations. Notable helper classes include the Texture class and AA_Render class, which contain attributes and objects related to texturing and anti-aliasing operations. All shading functions are called during the rastering stage of the pipeline.

Main: the main rendering loop of the shading program. Along with the rendering pipeline and function calls, Main utilizes the Streamlit framework on the front end to initialize a browser-based UI for the user to interact with.

UI Framework: Streamlit

For our project, we chose to use the Streamlit framework to turn our rendering script into a deployable web application. Streamlit was a perfect fit for our project because of its simple, but intuitive codebase and tutorials. We were able to easily create a sleek and modern front end, adding interactivity and allowing users to toggle features in our script on or off. With this framework, we were able to effortlessly rerun our rendering script on command from within the browser.

Once the scene and object files are read into the project, users can modify the scene file's properties, including the XYZ rotation of the 3d object as well as the type and texture of the object rendered.

There are presets that we made that we feel best show off the renderer. After running the selected preset, the user can edit any of the parameters they like to get their own unique image from our renderer. The main issue with the UI is that once you select a preset you have to submit to see what the preset sets the parameters too. I really wanted it to work like once you select the preset it sets the parameters automatically but had to settle with setting it after the submit button is pressed.

Optimizations

Compiling Python code into C code: Initially with Jupyter notebooks, the performance of our program was quite slow, clocking in at nearly 4 minutes. Because the Jupyter rendering environment did not compile Python code into a lower level language like C, we opted to move our code outside of the .ipynb environment (into .py and Visual Studio code) and compiled the code into C.

Saving repeat calculations into variables: To speed up the program, we saved the results of repetitive computation calls such as trunc() into variables and used those variables during color computations for a slight improvement in run time.

Manual calculation of the magnitude of vectors: We discovered that Python's numpy library is slower and has higher overhead on smaller vectors. By manually writing our own length() function, we cut the time spent on magnitude calculations in half.

Replacing Python image with the array version: We discovered that Python's getPixel function, while useful, slows down performance noticeably. As a result, we converted the Python images into their array variants and performed index lookups for the pixel colors, which cut the time spent on pixel lookups in half.

Normal Mapping:

We wanted the ability to have the crystal effect applied to any object and the key for this was to do it without modifying the geometry. So we decided to implement normal mapping with the vision that if we use a voronoi fracture normal map.
But since we were working on the renderer we built in class, the first step was to be able to convert objects from a standard 3D format to the json format used.

OBJ to .json

For our file format, we decided to use OBJ because its file structure made it easy to work with the geometric information within. The format first lists all vertices, with a "v" in front and the coordinates following it. We then read all of these coordinates in to create a collection of vertices.
Next, we handled the vertex normals suffixed by "vn" and vertex texture coordinates suffixed by "vt", and created collections for them as well. Finally, we parsed the faces, prefixed by "f". Faces specify the vertices in order as tuples: it contains the vertex index, normal index and texture

index. Since the .json format we used was based on triangles, we made sure to only pick triangulated meshes such that all the faces are triangles.

TNB matrix:

While researching normal maps, we realized that we could use the RGB values of a given image to nudge the normal of each face's local coordinate system. This made sense because the maps are invariant to transformations. We created a local coordinate system for a face, made up of its tangent, normal and binormal, aka the tangent coordinates, or the TNB coordinates. However, our renderer worked with normals in world space, so we needed a TNB matrix to convert the normal map normals from tangent space to world space, hence the usage of the TNB matrix.
The normal for a face is well understood but then our next question was, "what tangent do we choose?" There are infinite possible tangents on the face. Conventionally, we chose to check the texture coordinates of the current face and adjacent faces, and then use the normal that lies along their gradient. We then found the binormal by taking the cross product of the tangent and the normal.

TNB generator:

To achieve the TNB coordinates for all the faces of our object, we created another script. One challenge and quirk of the homework's json format is that it used averaged normals for the vertices. That is, if a vertex is present in multiple faces, the normal for that vertex will be the average of the normals for all the faces. This is what allowed us to do shading easily.
Knowing this, the script we created loads in a normal map texture and creates a hash map. This hash map then maps each vertex to the faces it is present in. Then for each face, it calculates the normal, the tangent and the binormal. Then, each vertex is assigned the average of these values from all the faces it is present in. The next issue we found was that the averaged tangent, normal and binormal did not form an orthogonal basis. To solve this, we applied something called the Gram-Schmidt orthogonalization to correct these and form a normal basis. Finally, we stored these values into our json format with additional keys, "tan" and "binorm" to represent the tangent and binormals respectively.

Normal Mapping:

Now with the TNB values in our json file and the TNB matrix needed to convert these values into world space, we had all the groundwork needed for normal mapping. While rendering each triangle, for each pixel, we performed a normal map lookup the same way we did a texture lookup. This gives us the RGB values of the normal map for that point. By using RGB values as the baseline for our XYZ "nudges" that the normal map would apply to the original object's normals. Since RGB values range from 0 to 255, we treated 128 as our default state indicating no nudge. Values greater than 128 were treated as a nudge in the positive direction and values less than 128 were a nudge in the negative direction. Based on this, we built the "nudge" normal from the RGB values of a texture. Because the normals supplied from the normal map are in tangent space, we interpolated the TBN values from all the 3 vertices to get the TBN values at

that point. This again caused them to be non-orthogonal, so we applied Gram-Schmidt orthogonalization and built the TBN matrix. Then multiplying the nudge normal with the TBN matrix and normalizing the resulting normal gave us the nudge normal in world space. To control the intensity of the normal map, we added a NORMAL_MAP_STRENGTH factor we multiplied our normal by to push out or push in the surface of our object. Finally, we added the nudge normal to the world space face normal and normalized the result to get the new face normal. This is used as the normal for all further lighting calculations and with that, we achieved normal mapping! With normal mapping, we essentially faked normals to pretend that each triangle has a much more complex terrain compared to its true surface. Because the normal of the object itself is changed, it lighting calculations react properly to these changes, giving us amazing detail and shading corresponding to these changed normals.

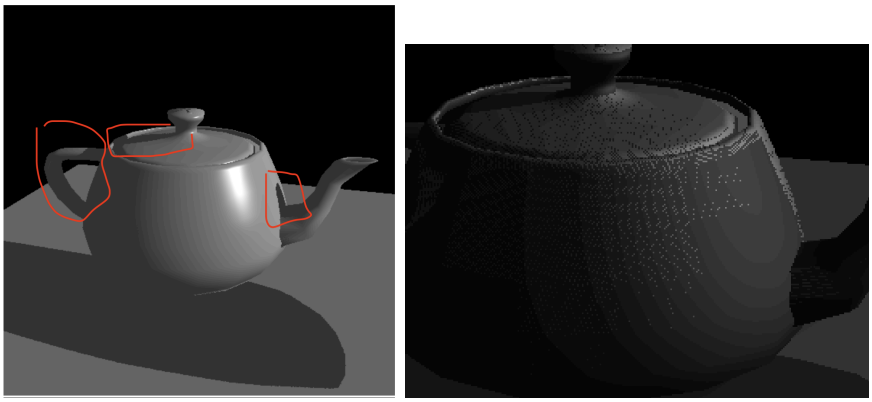
Voronoi Fractures as Normal Maps

Because we now had normal mapping, this allowed us to use Voronoi fractures as normal maps. The logic was that if we give each Voronoi crystal face a different color, the normal mapping would interpret these colors to allow each face to sit at a different angle, allowing each face to catch reflections from the lights and glimmer, giving us a great effect!

Shadow Mapping

To match the goal image we initially envisioned, we decided to add shadows for more realism. While implementing shadow mapping, we referenced 2 websites: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping and https://en.wikipedia.org/wiki/Shadow_mapping. From Wikipedia, we understood that to generate the shadow map, we needed to render the scene from the perspective of the light and record the closest z at each pixel, generating a depth map for this light. Given an object space point -> world space(objToWorld) -> light space (worldToLight) -> ndc (project+shift). The worldToLight matrix is calculated similarly to the camera but using the light direction. For the projection matrix, we used an orthogonal projection matrix because we are only using directional light. After all the transformations, the z depth was recorded into an image/map to use later for checking depth.

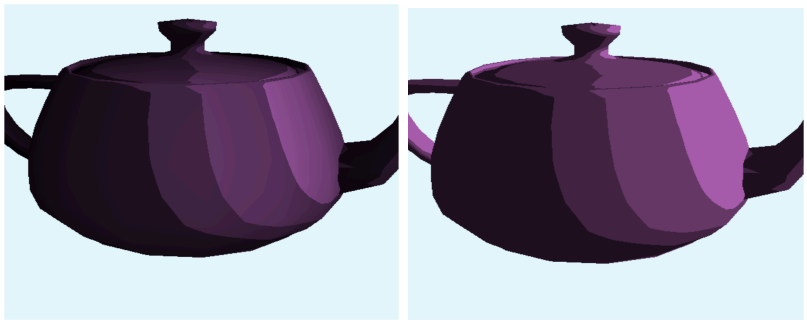
Once the shadow map is generated, we used it to determine the pixel color. With a given pixel, the object space vertex was used to check if the pixel is in shadow or not. The vertex is transformed into the light's NDC space, similar to how it was done in generating the shadow map. Once transformed, we check the current vertex's z value against the z value stored in the shadow map. If the current vertex z value was larger, then it was in shadow. If a pixel is a shadow, then we only gave it ambient lighting to show the effect of the shadow. If the pixel was not in shadow, we followed the regular color calculations.



Other things we considered was how to handle out of range errors and shadow acne (right image above). For out of range errors, we treated these as not in shadow and gave these get full light. For shadow acne, we followed the Opengl reference for shadow mapping and added a bias “based on the surface angle towards the light” by dot producing the normal and light direction to the z buffer check. In the end, we were able to produce shadows, but not all of them were correct. In the left image above, you can see the shadow on the floor doesn't match the shape of the teapot and isn’t positioned correctly. But if you look at the red marks, you can see a nice shadow between the nose and body of the teapot, the top of the teapot cap and the teapot handle being partly by the body.

Toon Shading

For toon shading, I referenced: https://www.lighthouse3d.com/tutorials/glsl-12-tutorial/toon-shader-version-ii/ and a paper **X-Toon: An Extended Toon Shader(2006).** The idea of toon shading is to find out how much light is hitting/lighting the object and instead of having a continuous range for the light intensity, we break it into discrete shades. We find the intensity by finding the dot product between the light direction and normal at the point. From the intensity value, I break it down into 4 shades where I modify the base color by a fixed amount, (you can see that in the right image below). Instead of the basic toon shading, I tried to implement X-Toon. The concept is to use depth as a parameter to determine the tone for the color within the discrete shades. So my idea is to use the z-depth, the farther away it is from the camera, the darker the tone and the closer it is then it matches the original shade. I first found the smaller and largest z-depth and used that to calculate my range from 0-1 and modify the shade by that value. This resulted in the left image below. In the image, you can see that the farther side of the teapot has become darker; however, darker parts all became too similar in color. If I had more time, I would have a “toon-texture” similar to the one in X-Toon to have control over the color tone. Instead of having the depth modify the color directly.



Perlin Noise

Perlin noise was quite straight forward. After finding Python's Noise library, we took some time to research how the library worked and how to apply the Perlin Noise values to our textures and normal maps. (It honestly made our crystal look worse.) I was hoping for some cool lighting effects due to the roughness from the Perlin Noise, but it just took away from the crystal effect. However, when we added in other textures, this is when perlin noise really got to shine, especially on the grass and marble presets. It took a lot of trial and error with the perlin noise values to make the final product look realistic and believable.

Background Images

It was a major challenge to get Anti Aliasing working with background images. The background images without anti-aliasing were quite easy, but when Anti Aliasing was applied, the background was set to black. When attempting to fix this, I tried to composite the object onto the background and use RGBA, but there ended up being a black circle between the background and the sphere. To fix this, we placed the background image in every filter image used for anti aliasing. Then, we follow the normal rendering procedure. It was a simple fix in the end but took me so many hours to figure out.

Results:

We are extremely happy with the results of our project and had a lot of fun working with UI and getting some very "unique" outputs. Our program is easy to install and run as a result of our UI framework and the default presets we've provided. Please mess around with it as much as you would like! Within our rendering program, using the presets will give the best results. However, you can easily adjust the rendering options, by scaling the textures, rotating the objects, and even rendering other objects and textures. While we are sad we could not get shadow mapping to work, we are collectively excited about what we have achieved and learned and hope you enjoy playing with our renderer.

Contributions:

Suvi: OBJ to .json conversion code, TBN matrix generation code and Normal Mapping
Carolyn: Base Renderer design, merged/ported separate code contributions into final renderer, Performance Optimizations, UI creation, and video script creation
Phillip: Shadow Mapping and Cel/Toon Shading, final project write up

Griffin: Perlin Noise, Support for Background Images, Anti-Aliasing and UI, handled setting up meetings, timelines, deadlines, video editing, and final submission.

We believe that everyone in the group contributed equally and enthusiastically.

References:

- Saty like cola bottle: https://www.turbosquid.com/3d-models/coca-cola-bottle-lowpoly-540827
- Stanford Bunny: raw.githubusercontent.com/harzival/stanford_bunny_obj_mtl_jpg/master/bunny.obj
- Tileable Voronoi Normal Map: https://www.filterforge.com/filters/11823-normal.html
- https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
- https://en.wikipedia.org/wiki/Shadow_mapping
- https://www.lighthouse3d.com/tutorials/glsl-12-tutorial/toon-shader-version-ii/
- X-Toon: An Extended Toon Shader(2006)