Experiment 1 Object Overlay Using 3D Models and 2D RGB Images:

Files Utilized:

- 3D Object Model: obj_000008.obj
- Camera Scene Information: camera.py and scene_camera.json
- Scene Ground Truth: scene_gt.json
- Test RGB 2D Image: 000005.png

Methodology:

Initial Extraction and Overlay

- Steps Extracted pose information from scene_gt.json and camera parameters from scene_camera.json.
- Rendered the 3D object model and aimed to overlay it on test image 000005.png.

Results and Analysis (First Part)

- Extracted pose information specifically for object ID 8 (obj_000008.obj), including a rotation matrix and a translation vector.

Challenges (First Part)

- Encountered installation issues with the pywavefront library for OBJ file parsing.

Solutions and Modifications (First Part)

Utilized alternative parsing methods to successfully extract object vertices and faces.

Further Steps:

- Applied a transformation matrix, derived from the pose, to the object's vertices.
- Projected the transformed vertices onto the image plane using camera parameters.

Results and Analysis (Second Part)

- Extracted camera parameters, notably the intrinsic matrix (cam_K).
- Employed original image dimensions (640x480 pixels) due to the lack of this information in scene_camera.json.

Challenges (Second Part)

- Projection and Alignment Discrepancies in the intrinsic matrix dimensions resulted in incorrect projections.
- Had to adjust the y-coordinates and translation for better alignment.
- Experienced issues with object scaling and orientation.

Sub-Challenges and Fixes

- **Flipping Orientation:** Corrected the object's orientation by flipping it along the appropriate axis.
- **Positioning and Alignment:** Made iterative adjustments for proper alignment.
- **Scaling:** Applied a scaling factor to match the size of the original object in the image.

Solutions and Modifications (Second Part)

- Reshaped the intrinsic matrix (K) to a 3x3 matrix, suitable for the proj_from_K function.

Final Outcomes:

- Projected vertices were normalized.
- Mapped the normalized coordinates to the original image's pixel coordinates.
- Successfully rendered the object, completing the overlay process.

Experiment 2
Trimesh Library for Integrating 3D Models onto 2D RGB Visuals

Files Utilized

- 3D Object Models: obj_000008.obj and obj_000008.ply
- Texture: obj_000008.png
- Test RGB 2D Image: 000005.png
- Camera Information: camera.py and scene_camera.json
- Ground Truth Information: scene_gt_info.json and scene_gt.json

Initial Setup and Data Exploration

- Loaded the 3D object models and the 2D RGB test image.
- Reviewed the camera and ground truth files to understand their structure and content.

Summary of Camera and Ground Truth Data:

- **Camera Data:** Includes camera intrinsic parameters (cam_K), rotation (cam_R_w2c), and translation (cam_t_w2c) from world to camera coordinates.
- **Ground Truth Data:** Provides object-specific rotation (cam_R_m2c) and translation (cam_t_m2c), as well as object ID (obj_id).

Methodology:

- Created a projection matrix and attempted to align the 3D object with the 2D image using the Trimesh library.

Results and Analysis

Challenges

- The Trimesh library limitations prevented rendering the transformed mesh, making it incompatible with the 2D RGB image perspective.

Solutions and Modifications

- Added material files (material.mtl and material_0.png) to enhance the object's appearance during rendering.
- Reloaded the 3D object with the new material and repeated the rendering and alignment process.

Further Steps

- Transform the 3D object using the previously calculated projection matrix.

Results and Analysis

- The 3D object with the applied material was successfully transformed.

Final Overlay and Rendering Needed

- Need to utilize additional tools like PyTorch3D and Blender for material-based rendering.

Custom Approach for Overlay

- Created a 2D rendering of the transformed mesh and overlaid it onto the 2D RGB image, but unsuccessfully applied the texture.

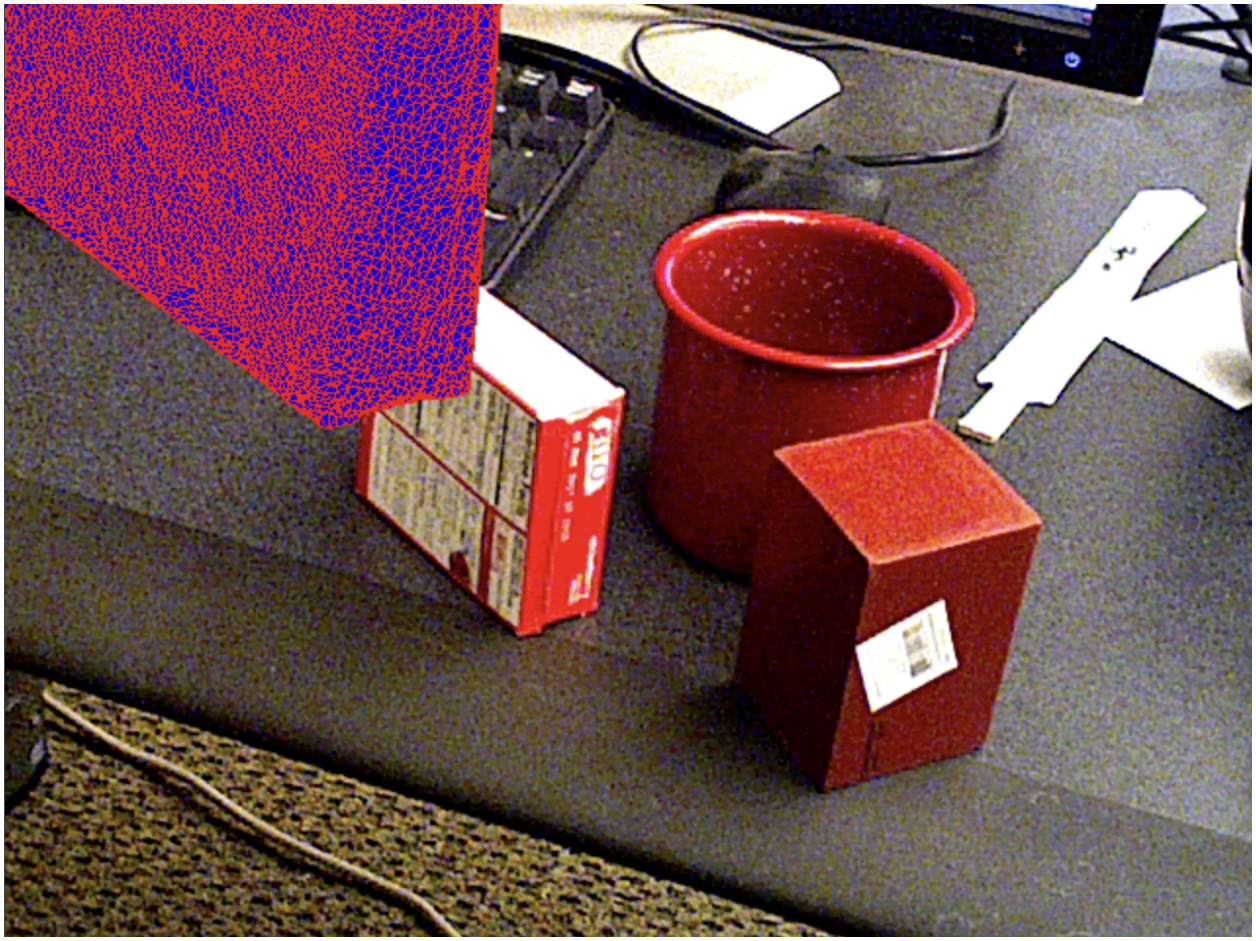
Final Outcomes:

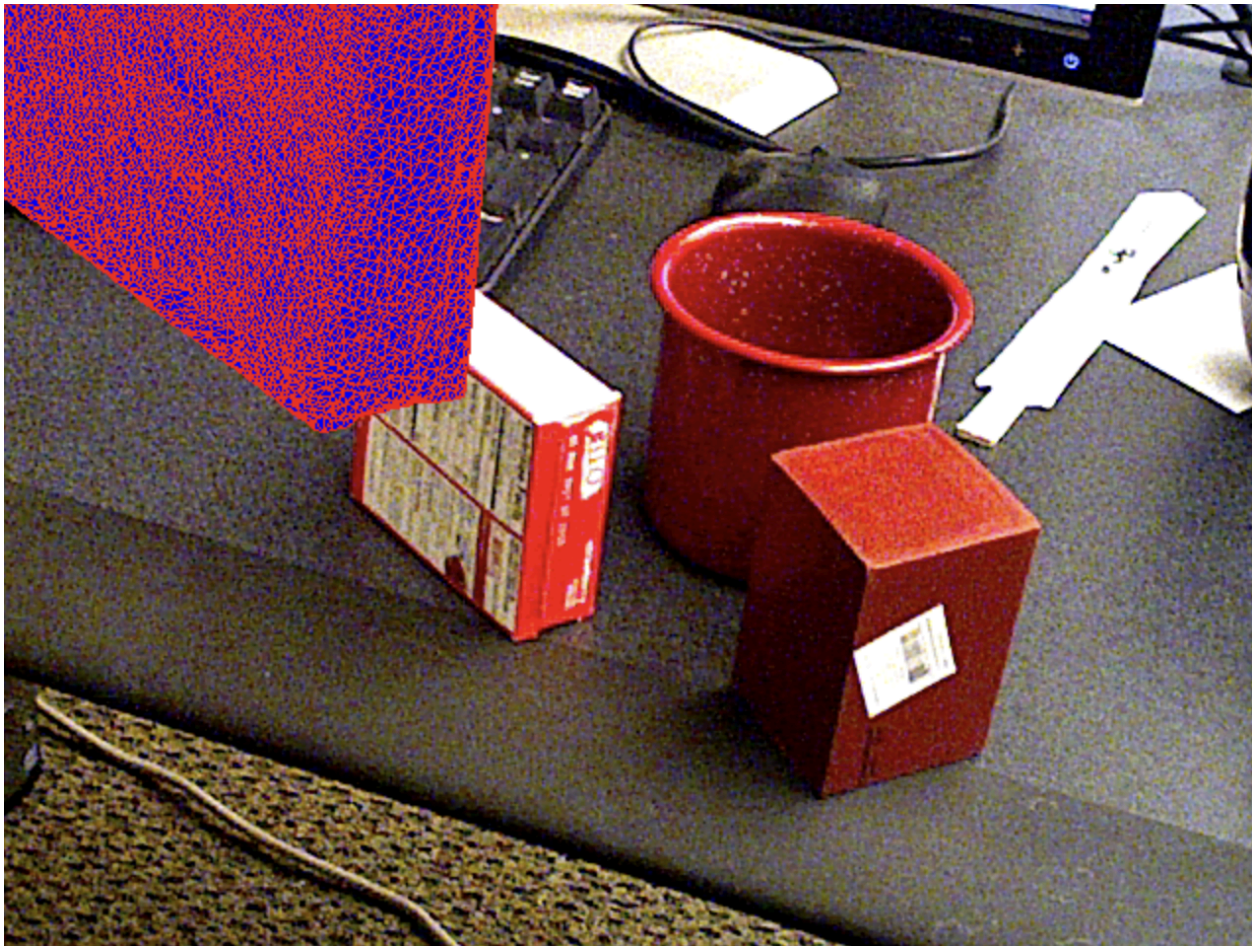
- The 2D rendering, represented by red dots, was successfully overlaid onto the RGB image, based on the transformed 3D object vertices, and aligned with the 2D RGB image perspective.

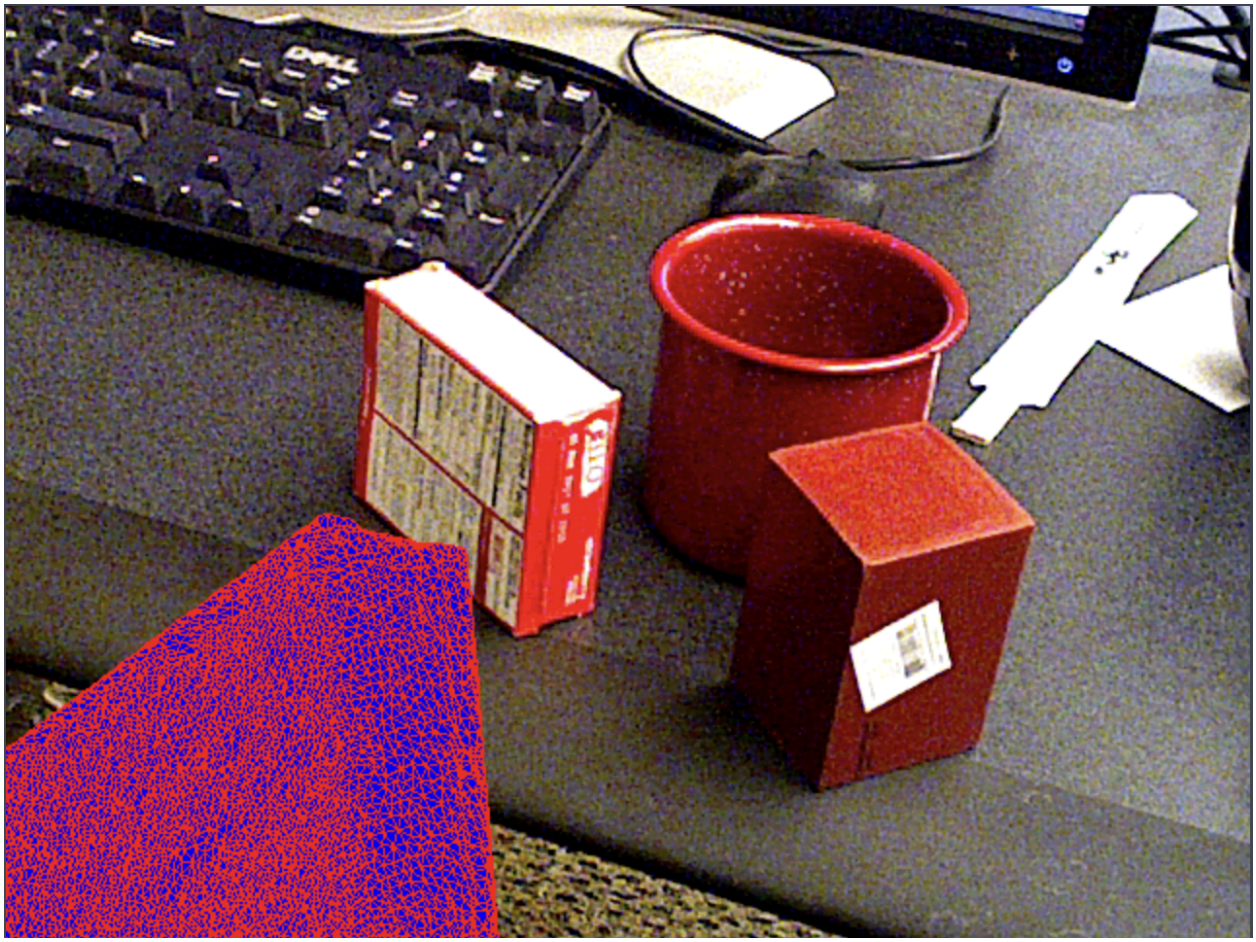
Next Steps:

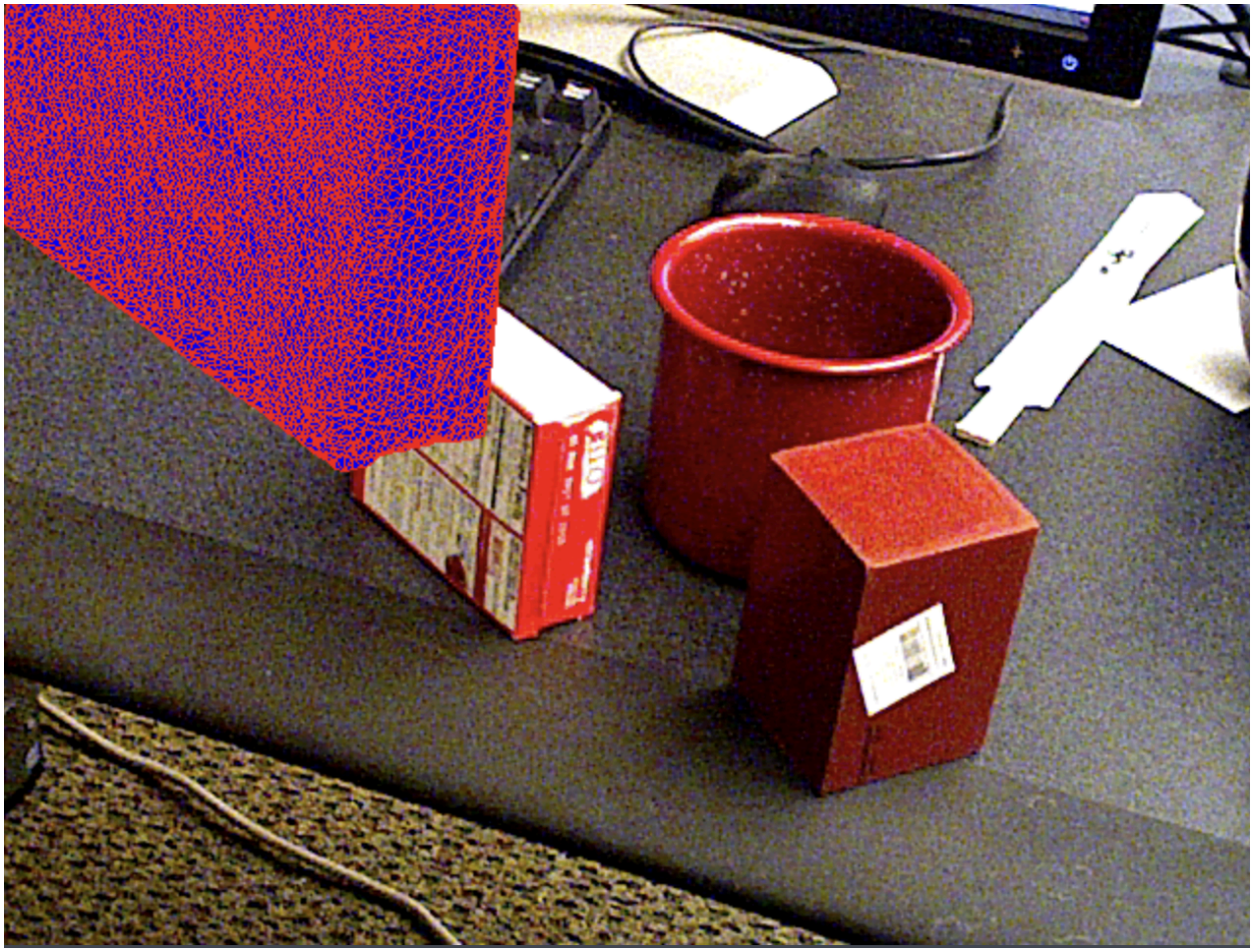
1- Learn Blender Python API for material-based rendering.
2- Continue Experiment 2 and complete a full rendering script for the 3D Models.
3- Keep following the Advice and Guidance of Dr. Tramblay and Professor Birchfield and learn from their feedback.

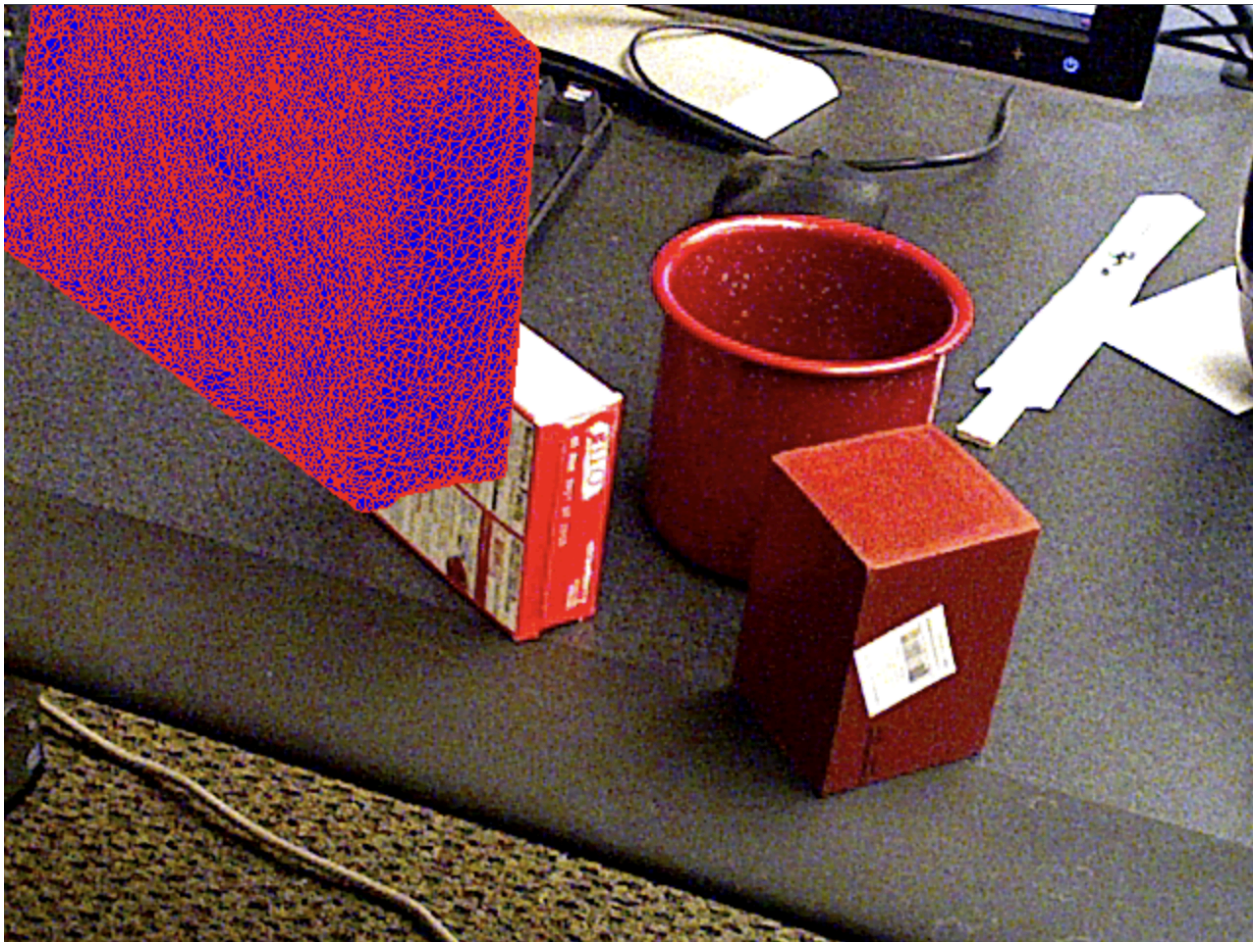
Results of Experiment 1:

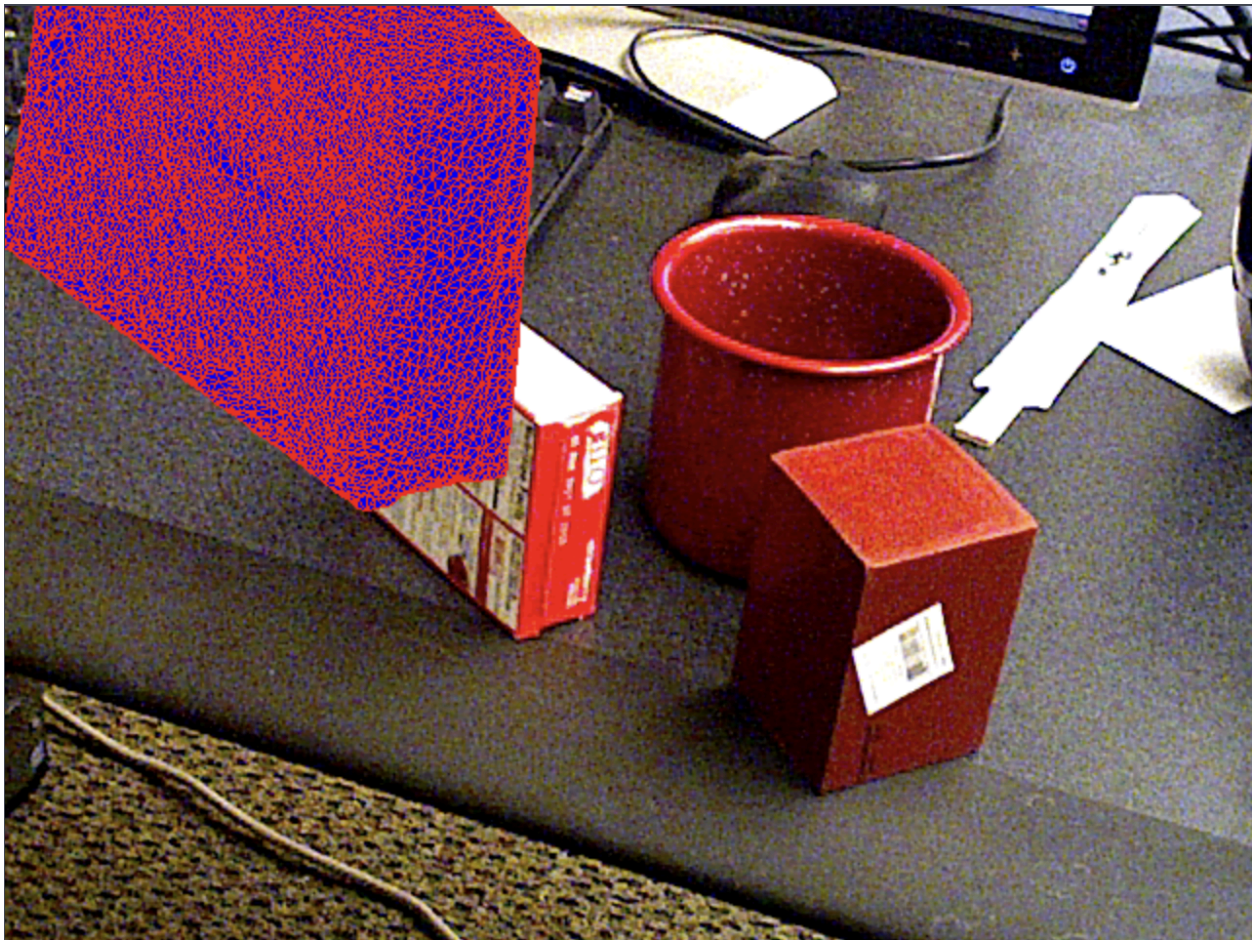


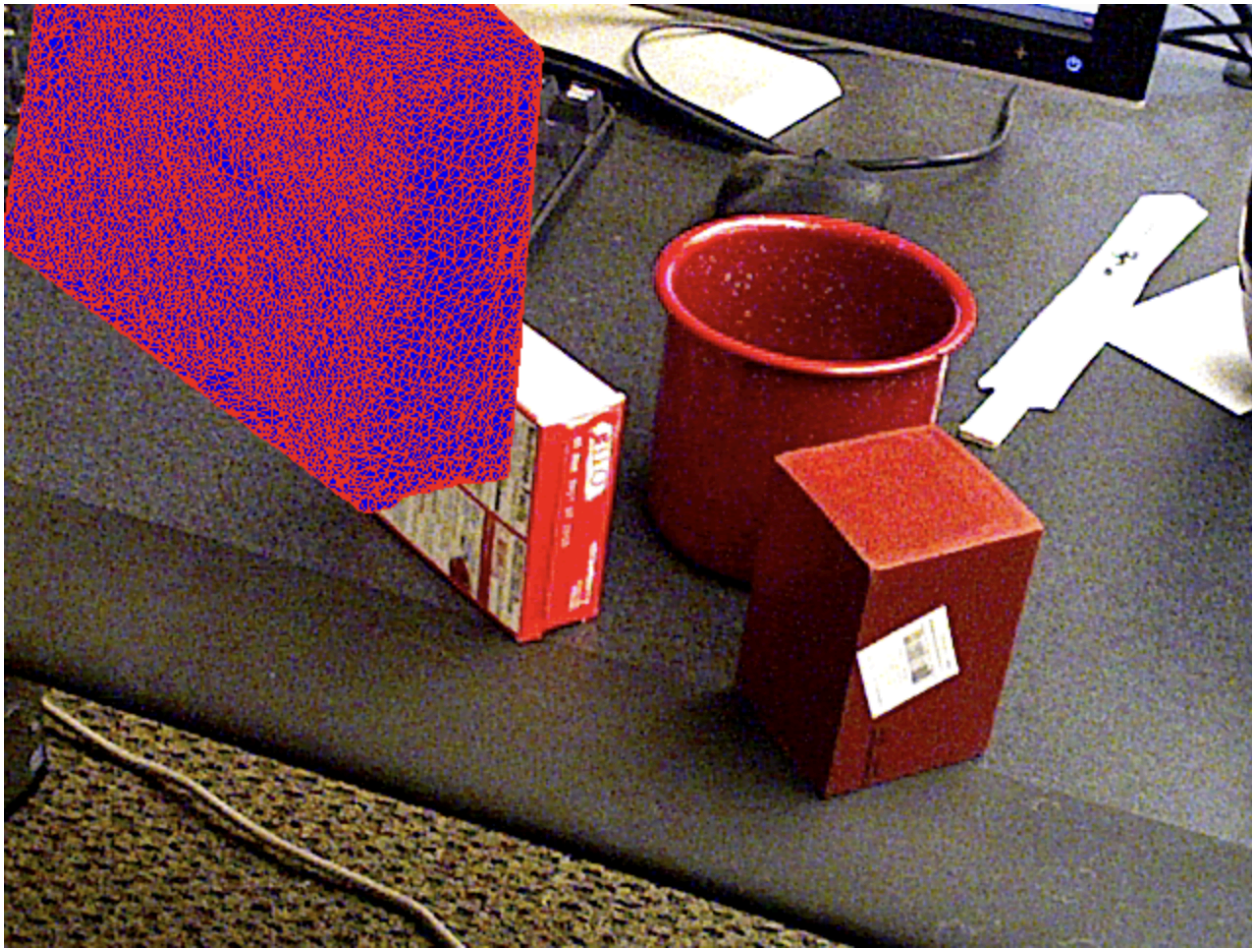


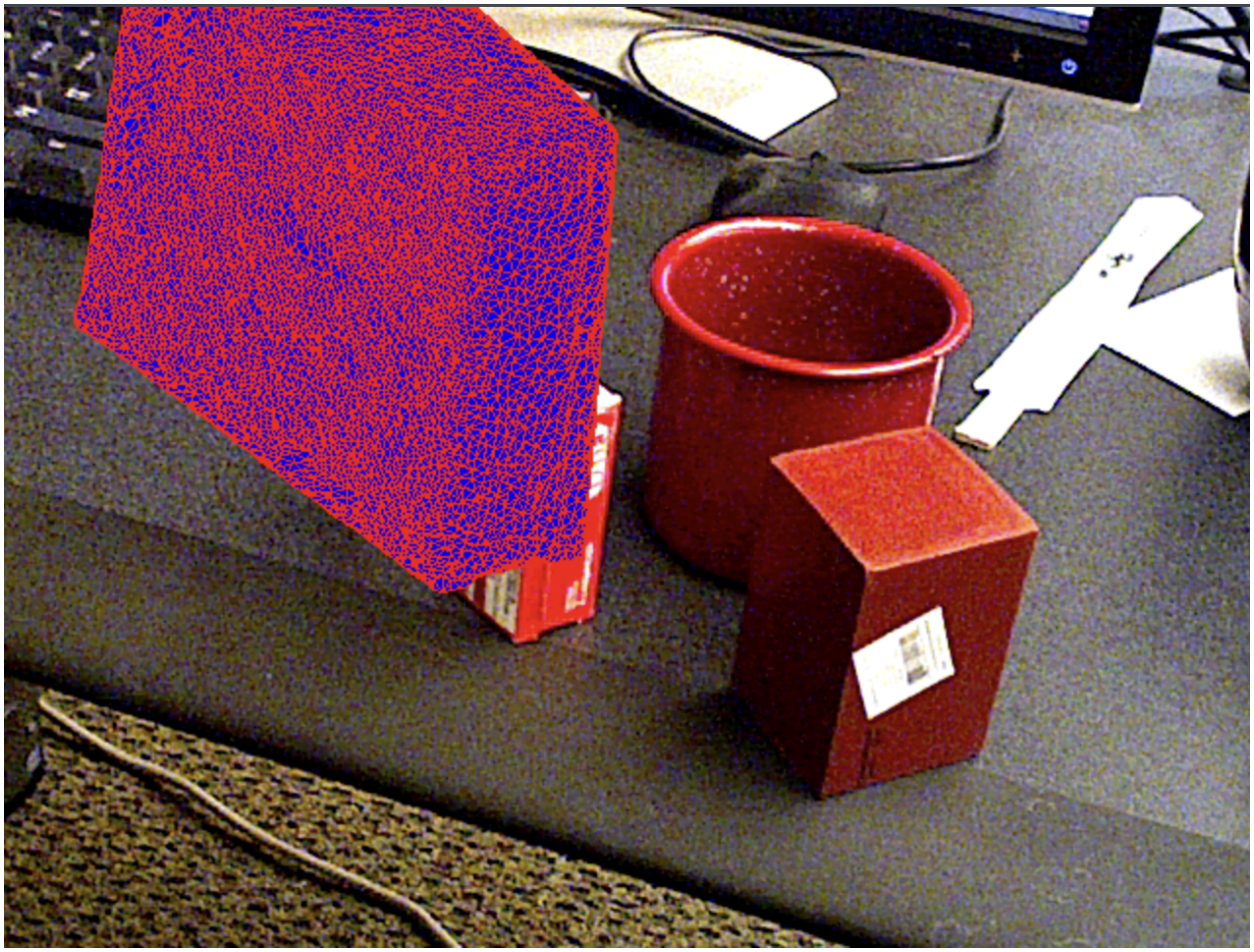


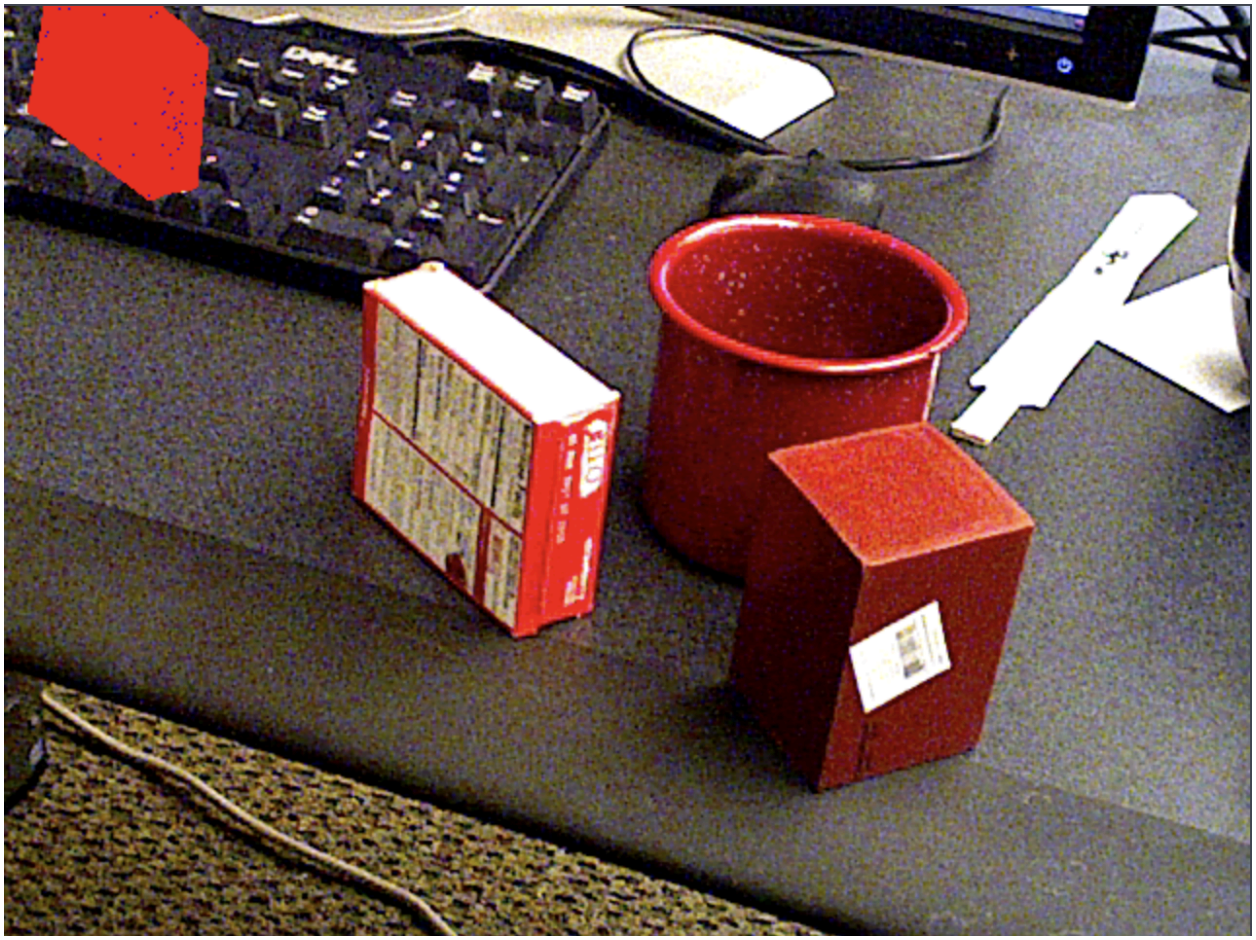


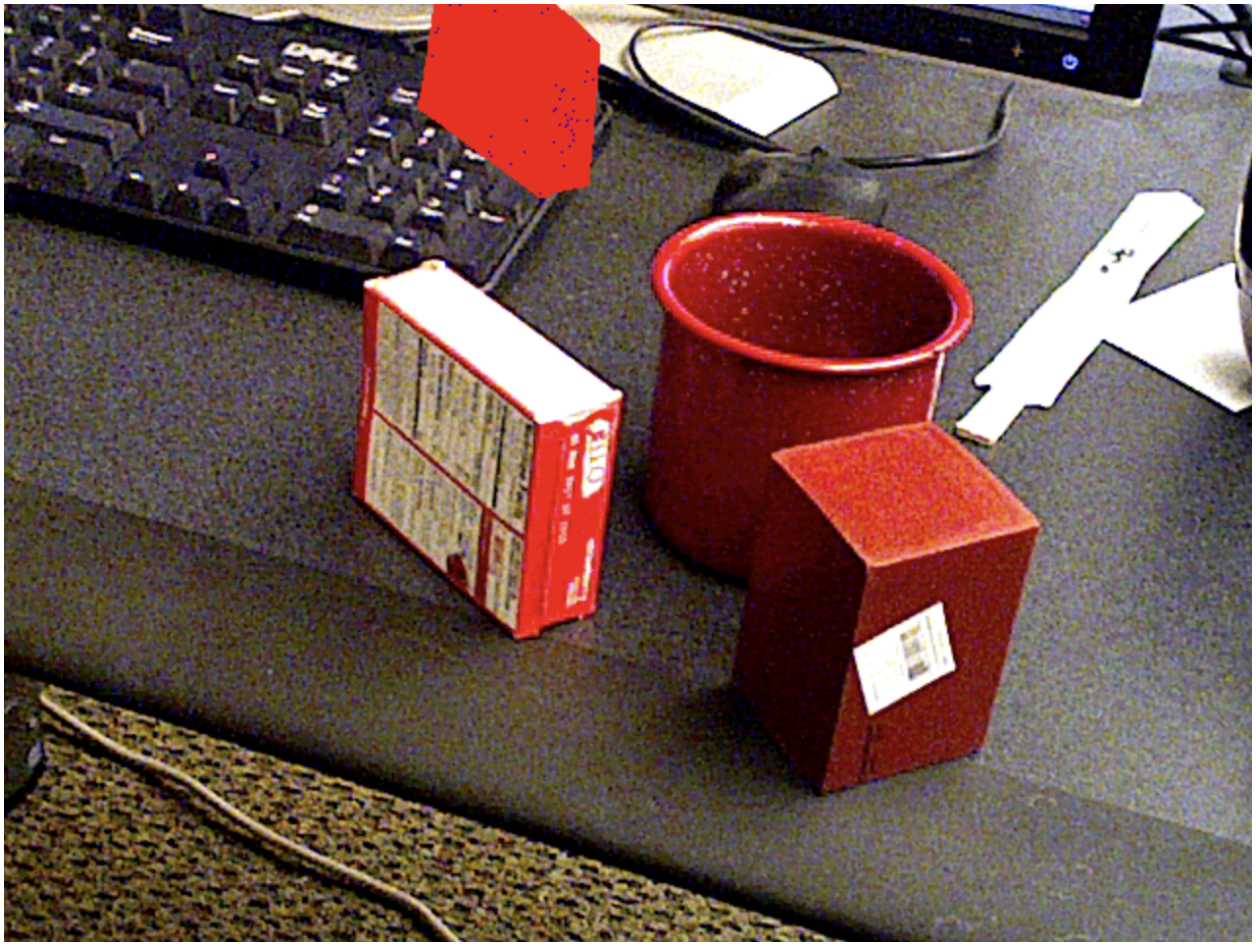


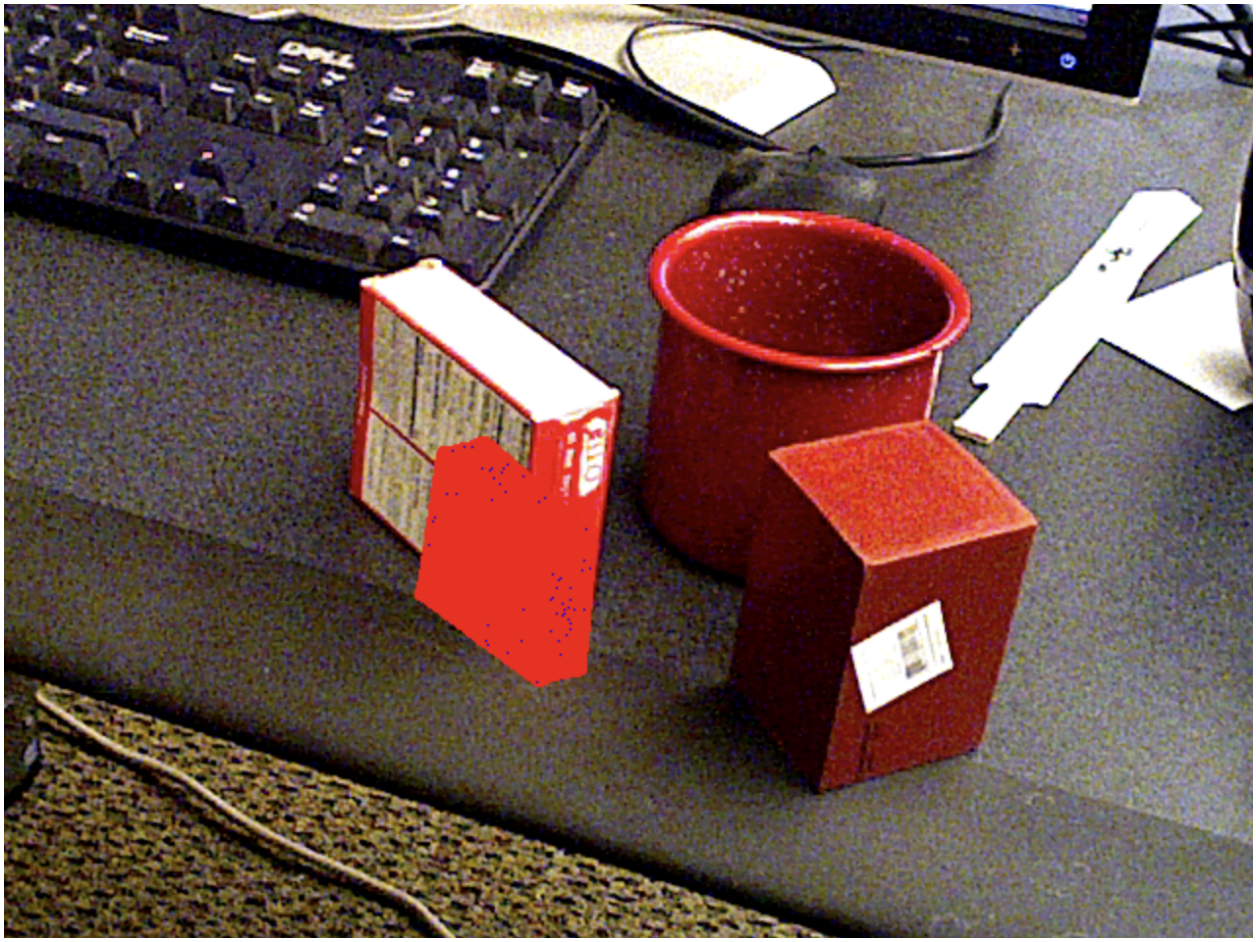


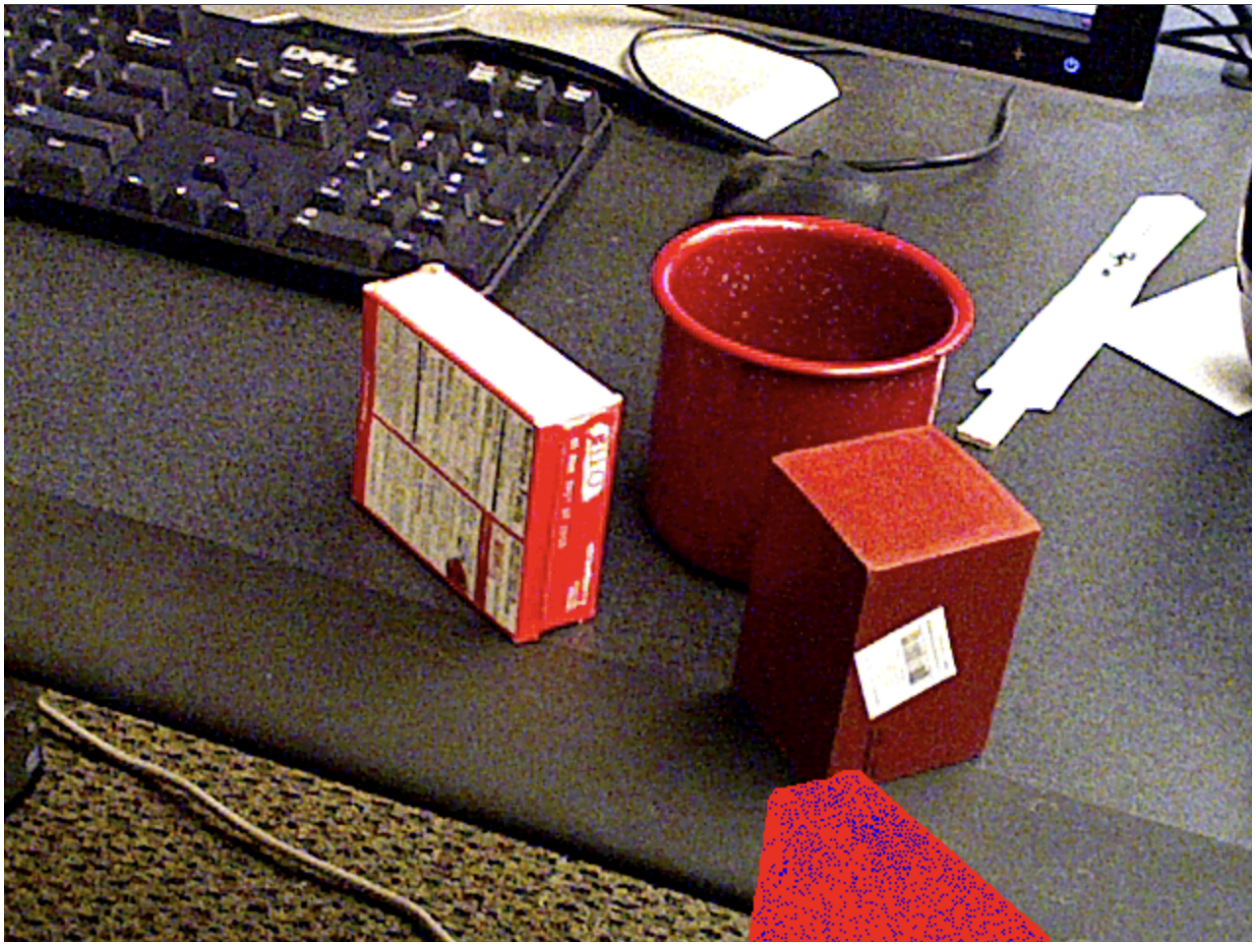


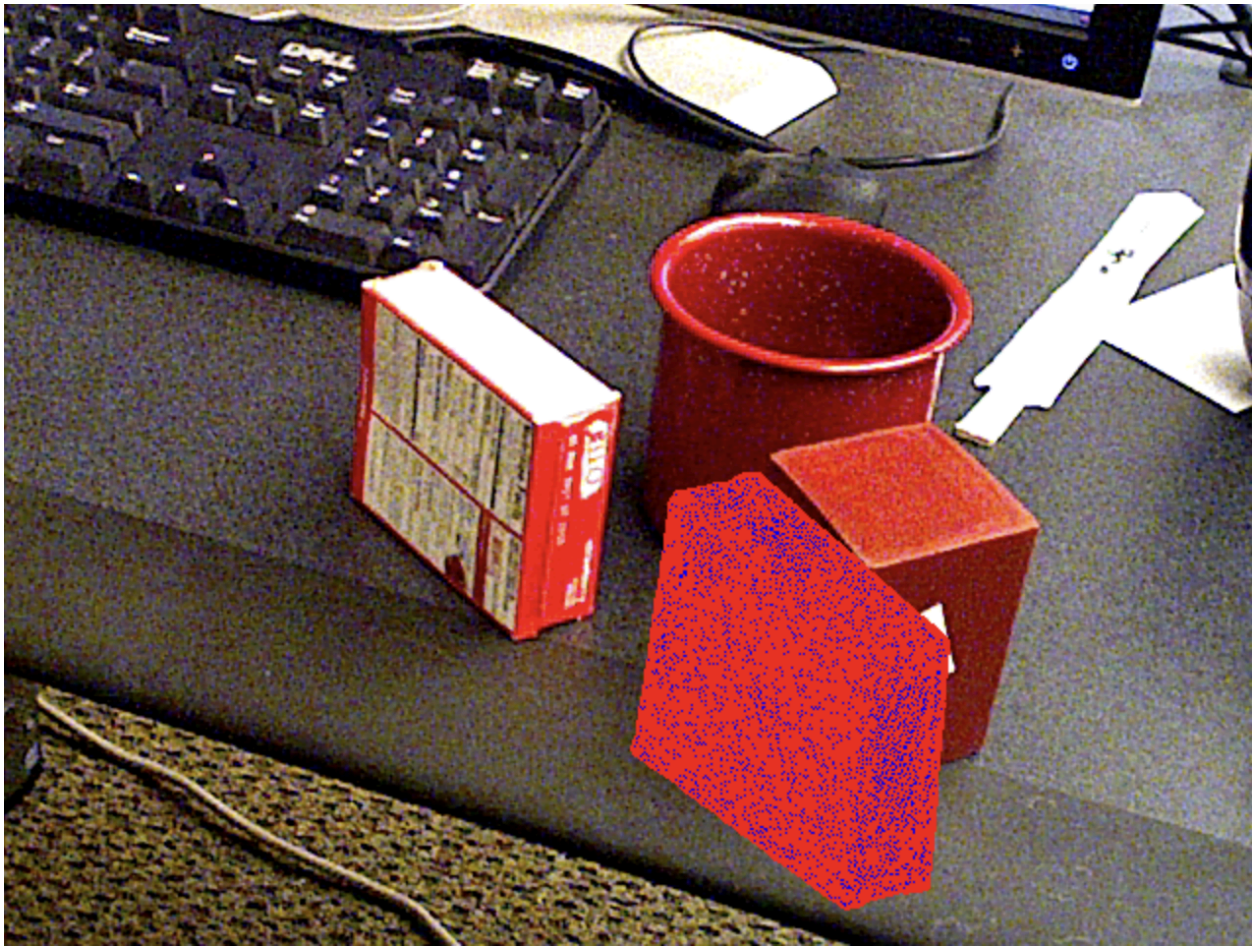


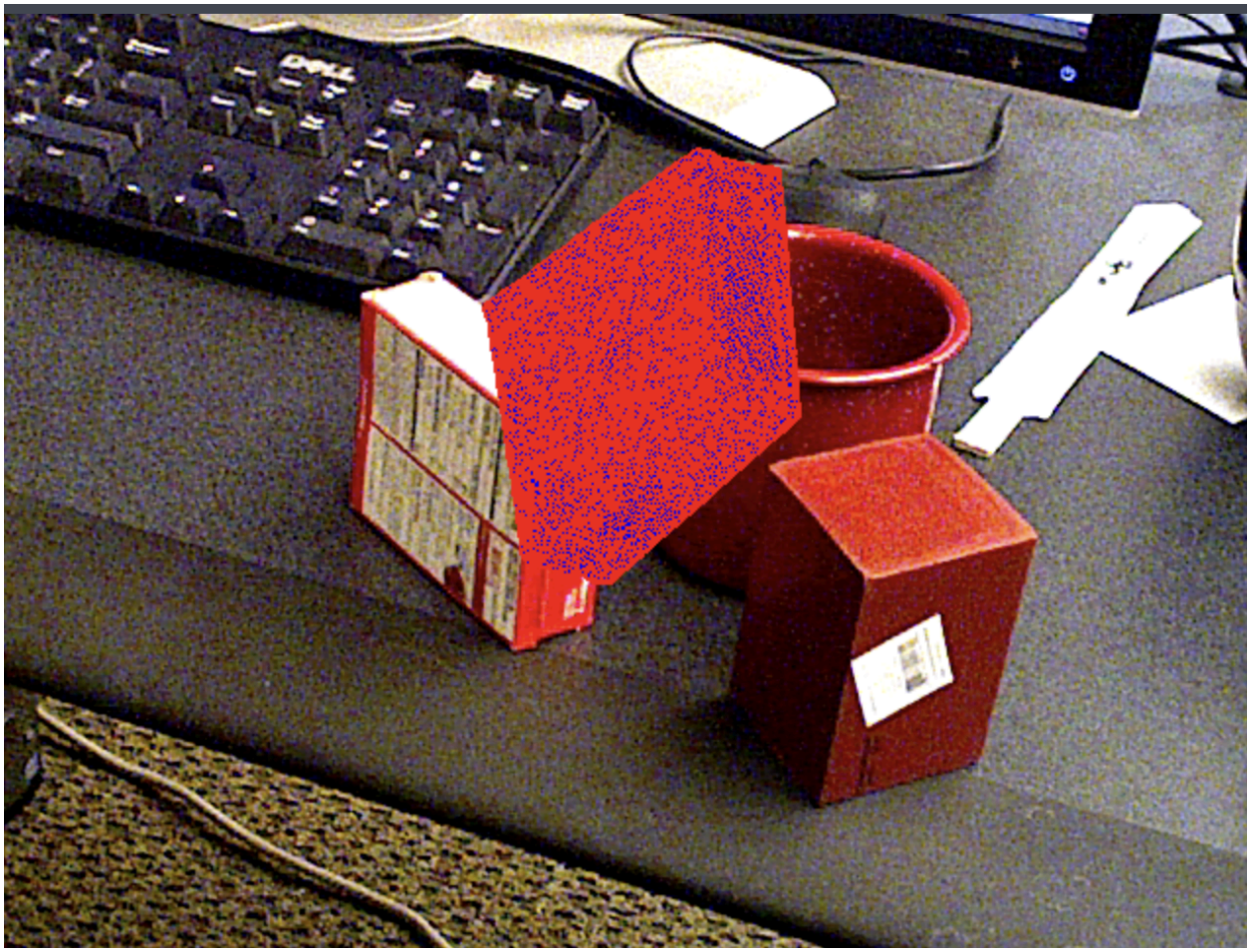

DELL

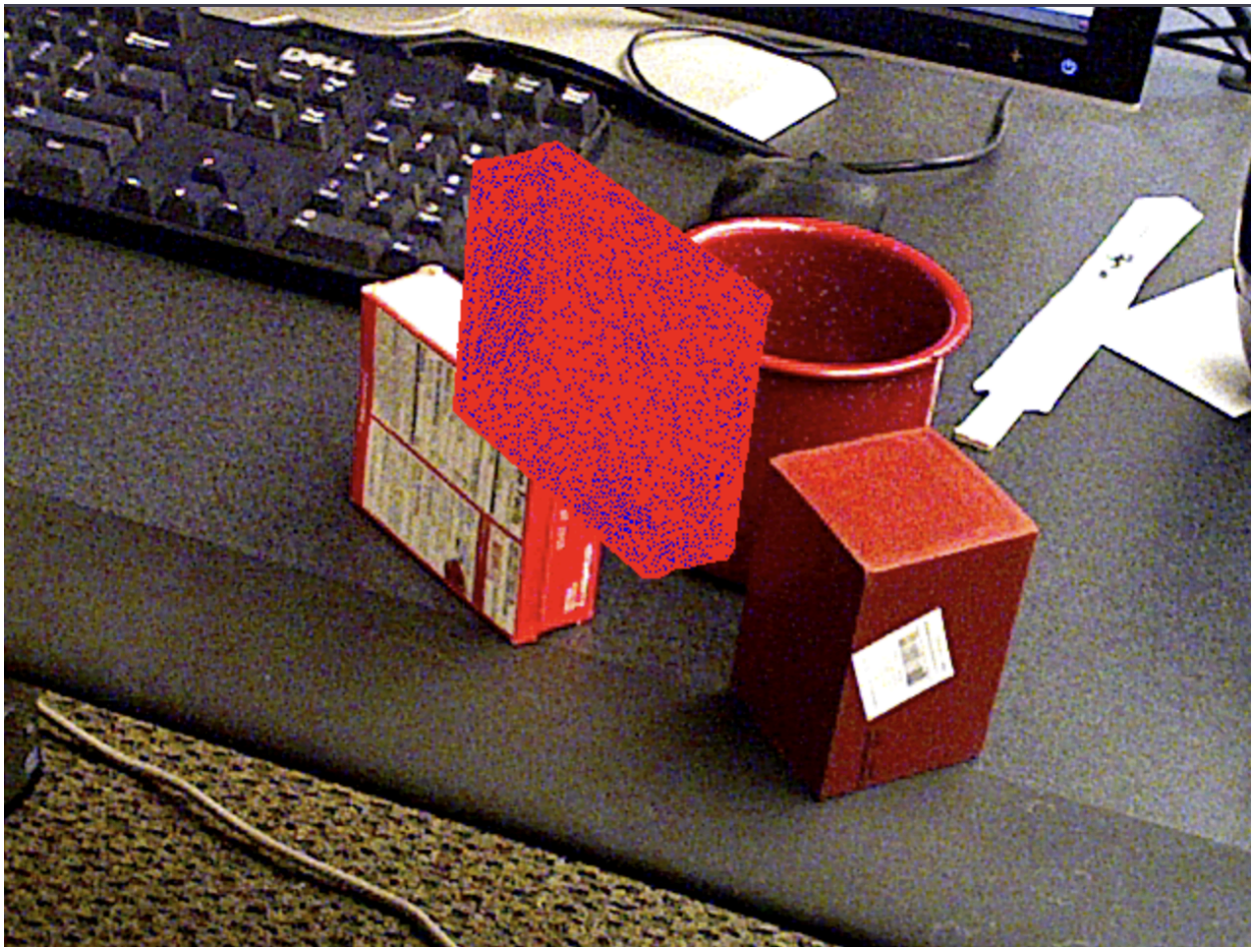

DELL

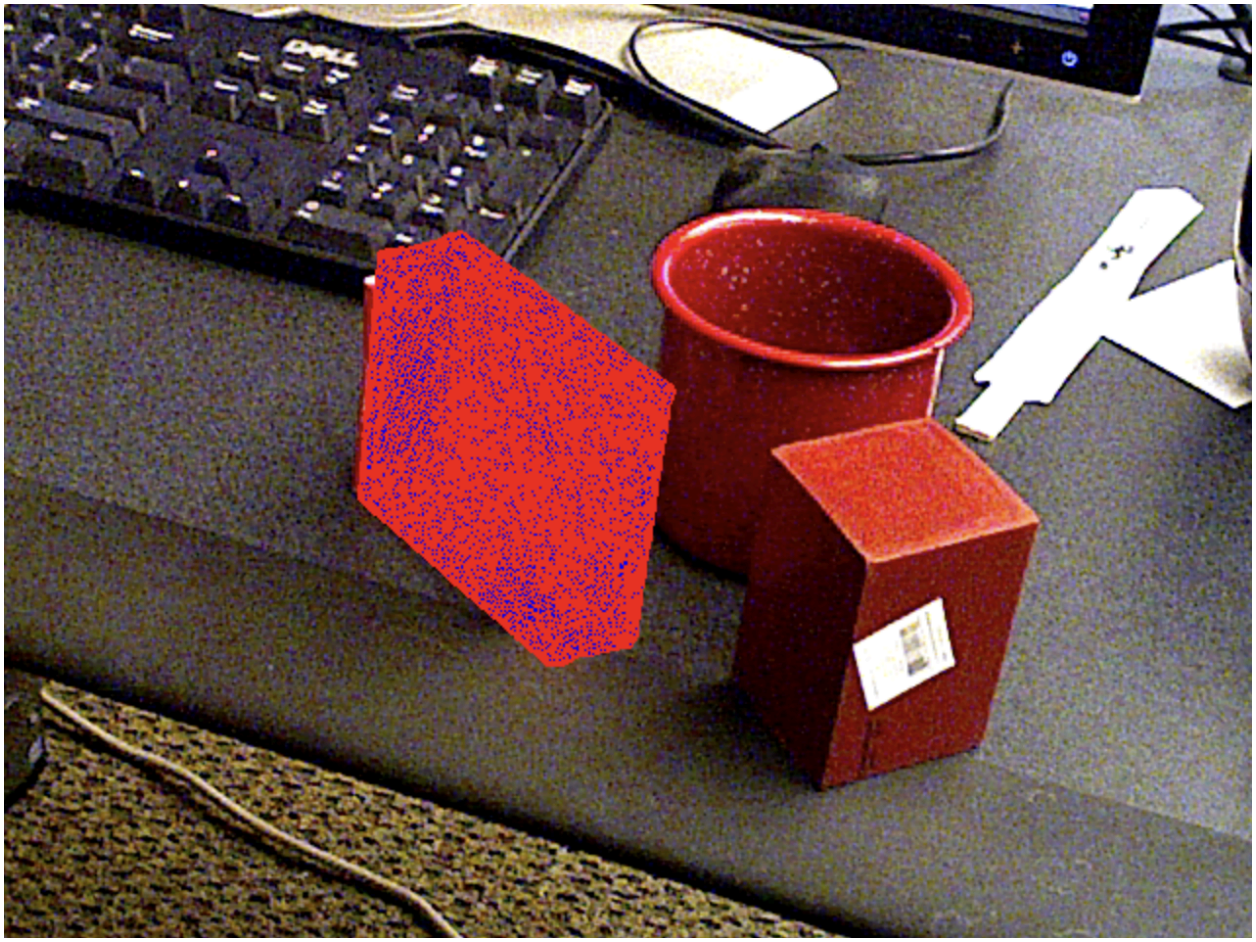

DELL


DELL

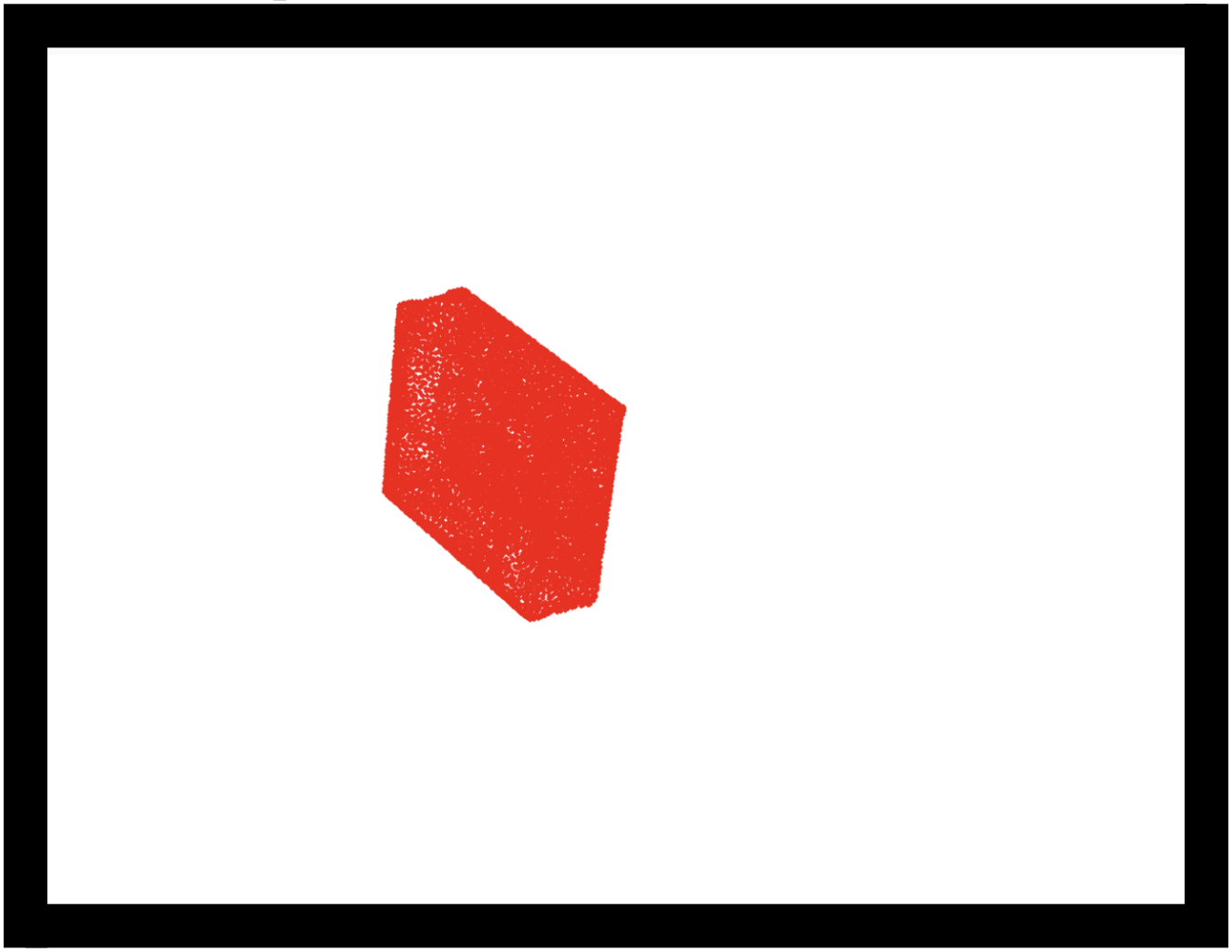

DELL



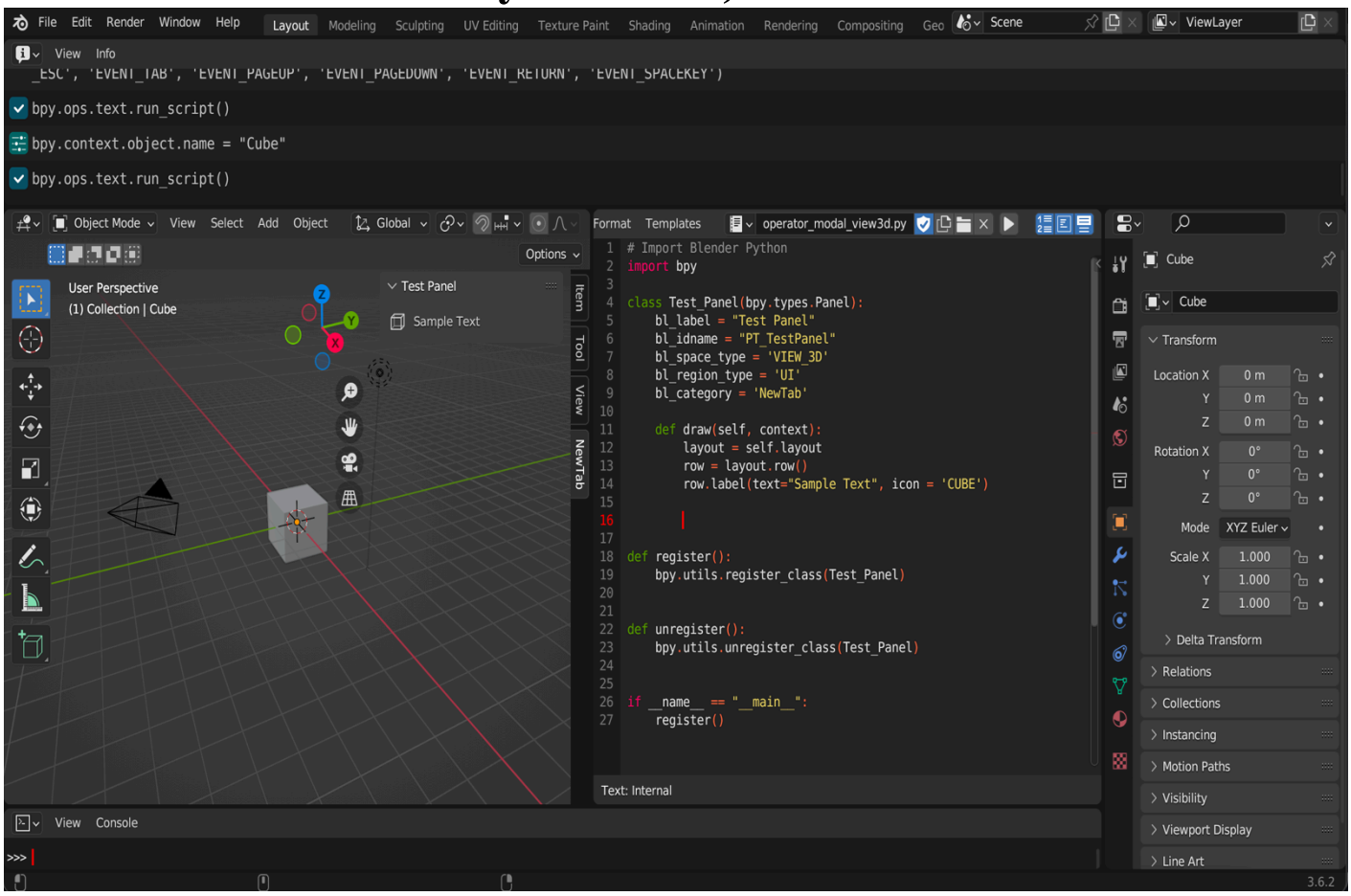

DELL



Results of Experiment 2:



Potential Experiment 3 (Still Learning how to use Blender and Blender Python API):



Experiment 3
Creating a add-on script for importing the texture correctly on to the models of the YCB-Dataset

The add-on is written in Python and uses Blender's API.

Import Statements and Dependencies
The code imports the following libraries:

- `bpy`: Blender's Python API for integration.
- `os`: Standard Python library for OS-level operations.
- `ImportHelper`: A helper class from `bpy_extras.io_utils` for file importing.
- `Operator`: A Blender class for custom operators.

Core Functions
`import_fun(context, filepath)`
This function is the heart of the add-on. It takes in the `context` and `filepath` as arguments. The function checks if the given file path exists and then performs the import operation.

Blender Operator Class
The code defines a custom Blender operator. This class is responsible for setting up the UI panel for the add-on in Blender. It inherits from `Operator` and `ImportHelper`.

Blender Panel Class
This class defines the Blender panel that contains the UI elements for triggering the import operation. It inherits from `bpy.types.Panel`.

Blender Add-on Registration
The script contains methods to register and unregister the add-on with Blender. These are crucial for the add-on to be recognized by the Blender application.

Demo and Code of the results are provided in the Google Drive Link:
https://drive.google.com/drive/folders/1x1Rmge7I5c9pkVNWcXzy69D1WXLEHEd3?usp=drive_link

References:
https://www.youtube.com/watch?v=pfhht_67x3k
https://blender.stackexchange.com/
https://docs.blender.org/api/current/index.html

Experiment 4
Automated Camera and Object Parameterization in Blender Using External JSON Data

Overview:
The code is written to manipulate camera and object settings in Blender. The code also reads JSON files to fetch camera and object parameters, which are then applied to the camera and object in Blender.

The code performs the following major tasks:

1. Initialization of Blender objects and scene parameters
2. Reading camera and object parameters from JSON files
3. Setting camera parameters in the Blender scene
4. Setting object parameters in the Blender scene
5. Inserting keyframes for camera and object to enable animation

Detailed Analysis

Initialization
The code initializes Blender's camera and object settings, along with rendering resolutions. It sets the camera sensor width and height, and also identifies the object based on its name.

Reading JSON Files
The code reads two JSON files: one containing camera settings (`scene_camera_path`), and another containing object settings (`scene_gt_path`). These JSON files are read into Python dictionaries for later use.

Setting Camera Parameters
The code sets the camera's focal length in millimeters. This is calculated using the focal length in pixels, sensor width, and rendering resolution. The camera's rotation and translation matrices are also set.

Setting Object Parameters
The object's rotation and translation matrices are set based on the data read from the JSON file. These settings are applied to align the object within the global coordinates of the Blender scene.

Inserting Keyframes
Keyframes are inserted for both the camera and the object. This is done to enable animation within the Blender scene. Keyframes are inserted for location and rotation parameters.

Conclusion

The Python code effectively manipulates Blender's camera and object settings based on external JSON files. It sets various parameters like focal length, rotation, and translation matrices and also enables animation through the insertion of keyframes.

Demo and Code of the results are provided in the Google Drive Link:

https://drive.google.com/drive/folders/1YDJFqTc9BzkChSIqYj5DN8FMNQGuoxOh?usp=sharing

References:

https://www.youtube.com/watch?v=3Q5L1DZqBQw

https://www.youtube.com/watch?v=pfhht_67x3k

https://docs.blender.org/api/current/bpy.types.Camera.html

https://blender.stackexchange.com/questions/23433/how-to-assign-a-new-material-to-an-object-in-the-scene-from-python

https://realpython.com/python-json/

https://blender.stackexchange.com/questions/7358/python-performance-with-blender-operators