INDIAN INSTITUTE OF TECHNOLOGY DELHI

DEPTT./CENTRE: **COMPUTER SCIENCE AND ENGINEERING**

REPORT OF THE PROJECT SHORTLISTED UNDER **SURA - 2015**

Project Titled: **Metal Corrosion Rendering**

Submitted by:

Name of Student	**Department**	**Entry No.**	**Email (Contact no.)**	**Signature**
Rishit Sanmukhani	CSE	2013CS10255	cs1130255@iitd.ac.in (9711932719)	
Yash Gupta	CSE	2013CS10302	cs1130302@iitd.ac.in (8588824291)	

.

Name and signatures of the Facilitators: _____

Prof. Subodh Kumar

Department: Computer Science Engineering

Mobile no. of the Facilitator: _____

Date of submission of report in IRD:

Table of Contents

Acknowledgements

We would like to thank Prof. Subodh Kumar for advising us enthusiastically throughout the course of the Summer Undergraduate Research Project (SURA). We would also like to thank him for giving us this opportunity to work on this exciting project, and for encouraging us at all times. We would also like to thank Nisha Jain (Research scholar) for providing key insights throughout the course of the project.

Rishit Sanmukhani (2013CS10255)
Yash Gupta (2013CS10302)

Chapter 1: Introduction

One of the key areas in the field of computer graphics is realistic rendering of corroded objects. There are several approaches including physical modelling and empirical models. Physical modelling methods usually require a lot of computation resources and are not viable for real-time results. Therefore, we take the empirical modelling approach. The focus of this project is to produce realistic renders of corroded metal objects using photographs of sample corroded surfaces.

The main challenges are:

- Identify important features of the surface from the photographed image
- Extracting those features with high level user input
- Creating textures for the object mesh according to it
- Rendering the object mesh with generated textures

The rest of this document is structure as follows: Chapter 2 discusses the methods we used for various objectives of our projects, Chapter 3 shows and compares the result of our method with previously used methods and finally we derive conclusions for further improvements in our methods.

Chapter 2: Methodology

Identification of parameters

First step in the rendering pipeline was to identify the parameters representing appearance of corroded objects. Parameters like diffuse color, metallicity, smoothness, bumpiness, depth information and reflectance play a major role in rendering of object model. For example, diffuse color will determine the final color of rendered object model, bumpiness will determine irregularities at the surface level and depth information will give displacement in the polygon mesh representation of the object model. This makes parameter selection an important step.

We decided to use Unity3d - 5 for rendering the object model. Motivation for using Unity3d-5 was because it has many built-in shaders and parameters of those built-in shaders can be changed. This saves a lot of writing custom shaders, as we can use Unity3d built-in shaders as base for our rendering purposes. Unity3D also provides means to customize the pipeline through scripting and shader configuration files.

Following two built-in shaders offered us best quality, control over parameters as well as real-time performance:

1. Standard Shader with metallic setup
2. Standard Shader with Specular setup

Both of above two shaders were major addition in Unity3d-5 edition replacing large number of old built in shaders. Our main objective was to render metallic objects in corroded state, so we decided to use Standard Shader with metallic setup. It gave us right set of parameters for our object model.

We identified and worked on following parameters offered by standard shader with metallic setup:

1. Albedo

   Albedo parameter offers you control over base color of the surface. We had an option of providing single color or providing a texture map for our Albedo parameter. Later option gives us more control over appearance of surface of final rendered object model. We call the texture map assigned to Albedo parameter as Albedo Map.

2. Normal Map

   Normal map offers control over per-pixel surface details such as bumps and grooves. We have to provide a texture map (of type normal map) representing fine surface details of the object model. It is important to mention here that normal map does not alter the polygon mesh geometry of the object model, rather it just modifies the normals as per the texture values. This modification of normals determines how lightning is calculated on the surface and thus giving appearance of embossed/engraved surface details.
   We use Unity3d to convert our greyscale depth map to normal map. We call our greyscale texture as Depth Map.

3. Metallic

   Metallic characteristics are modified by the following two parameters:

   a. Metallicity

      This parameter defines how "metal-like" the surface is. More is the metallicity, more are the reflections from the surrounding making albedo color less visible.

   b. Smoothness

      This parameter defines the smoothness of the surface. Smoother the surface, lesser is the deviation of micro-facet normals from given surface normals and such a uniform surface reflects light in a narrowly focused area. Because of it reflections of objects in environment are clearly visible. But if smoothness level is very low, then light is reflected in a diffuse fashion.

   Thus both of the above parameters alter the reflectivity and light response of the surface thereby modifying metallic nature of the surface. We have the option of assigning a uniform value of metallicity and smoothness to entire object model or use a texture map. We prefer using texture map as it gives helps in capturing the variation of metallic nature of object model. In the texture we use, Red channel is used as metallicity and alpha channel is used as smoothness. (This is as per Unity3d convention)
   We will call this texture map as Metallic Map.

There are number of other parameters which we could have worked upon, but in this project we considered using above four parameters. This gives us sufficient variation in capturing the appearance of the object model.

Due to corrosion, some part of the metallic object gets removed and thus geometry of the object changes. If we consider polygon mesh representation of the object model, depending upon

corrosion degree, voxel of this mesh gets removed due to corrosion. To capture this, we need to change to geometry of the object model depending on the corrosion degree and none of the above mentioned parameters alter the geometry of the object. So we generate a texture map which gives us amount by which geometry of the object will get altered. We call this texture map as Displacement Map.

Following is the pseudo code of script which uses Displacement Map and modifies mesh geometry:

```
Texture2D tex = load Texture
Mesh mesh = load Mesh
float factor = -0.5f
for (int i=0 ; i<mesh.vertices.length ; i++)
       mesh.vertices[i] +=
              mesh.normals[i] * tex.value(mesh.vertices[i].uv) * factor;
mesh.recalculateNormals();
```

Extraction of Parameters

We have identified following parameters to represent appearance of object model:

1. Albedo Map
2. Depth Map
3. Metallic Map
4. Displacement Map

We introduce a “weathering map” which represents the corrosion degree of the input object’s surface. Each pixel of weathering map will have a value between 0 and 1 (0 has least corrosion degree). Corresponding to the weathering map, we need to generate above four maps to render the object in Unity3d. In order to generate above four maps, we need to have information about the values of these parameters at different corrosion degrees. In this part of the project, we try to extract this information from a photograph of sample corroded surface. We restrict ourselves to extraction of “Albedo value” from the photograph to limit the scope of this project.

Algorithm (*colorgen*)

Given a photograph of sample corroded surface (with shadows and specular highlights removed with existing techniques, if possible), we present following technique for extraction of Corrosion-Albedo (Color) mapping (in the form of a gradient) from it:

1. User interaction

   Our methods requires a set of points (pixel coordinates) in the image with their corrosion degree from user (ordered triplets, $\boldsymbol{(x_i, y_i, deg_i)}$). These set of points are called control points. Control points gives us the idea about color of the object at particular corrosion degree. Number of control points are not fixed and user can provide as many as he wish. In general, higher the number of control points, better the generated color gradient.

2. Generating graph

   We determine the unique colors present in the image. We sample down the colors by considering all colors component-wise within *EPSILON* of a given color as identical to reduce the number of colors (thus computation time in the subsequent steps). Unique colors calculated in this step forms the nodes of a graph ***G(V,E)***.

   In order to decide the neighbors of a node in the graph, we calculate distance of each node to given node and if it is within a set threshold (*NEIGHBOUR_THRESHOLD*), we add an edge between the nodes with weight as the distance we just calculated. This distance is a distance metric b/w the colors. We use the Euclidean distance between colors in some color space (RGB, YUV or L^*ab for this project; configurable). For each node in graph, we take the nearest *MAX_NEIGHBOUR* neighbors only, and remove others.

3. Generating corrosion-albedo mapping

   We find the nodes $\boldsymbol{c_i}$ corresponding to the given control points $\boldsymbol{(x_i, y_i, deg_i)}$ in $\boldsymbol{V}$, and assign the corrosion degree $\boldsymbol{deg_i}$ to $\boldsymbol{c_i}$. These values are the basis for assigning corrosion degrees to other nodes in $\boldsymbol{V}$.

   Now, considering each $\boldsymbol{c_i}$ as source node, we calculate single source shortest path distance to all other nodes in $\boldsymbol{V}$ using Dijkstra's algorithm on $\boldsymbol{G}$. For each node $\boldsymbol{n_j}$ in $\boldsymbol{V}$, we have the shortest-path distance $\boldsymbol{d_{ij}}$ to each $\boldsymbol{c_i}$. We then calculate a weighted sum of corrosion degrees ($\boldsymbol{deg_i}$) of the control-point nodes ($\boldsymbol{c_i}$) based on the distance to respective control-point node ($\boldsymbol{d_{ij}}$) to calculate $\boldsymbol{deg_j}$. We use weights inversely proportional to $\boldsymbol{d_{ij}}$. This gives an

effect of corrosion degree leaking around the control-point node, with each control-point node maintaining its value as given by the user.

For some $\boldsymbol{n_j}$ in $\boldsymbol{V}$, there may be some $\boldsymbol{c_i}$ which has $\boldsymbol{d_{ij}} = \infty$. This means that $\boldsymbol{c_i}$ and $\boldsymbol{n_j}$ lie in different connected components of the graph (i.e. they are disconnected). In this case, $\boldsymbol{deg_j}$ depends only on the $\boldsymbol{c_i}$'s which lie in $\boldsymbol{n_j}$'s connected component in $\boldsymbol{G}$. This results in better results as we do not try to explicitly relate control-points which do not have a gradual gradient to each-other in the colors of the image. The level of connected-ness is decided by the *NEIGHBOUR_THRESHOLD* and *MAX_NEIGHBOURS* along with distance metric we use for colors.

After the last step, we have $\boldsymbol{deg_j}$ for each $\boldsymbol{n_j}$ in $\boldsymbol{G}$. Now we calculate inverse mapping from corrosion-degree (down-sampled to the number of levels we want in output gradient) to color value of a node. This results in a one-to-many mapping, with multiple colors for each corrosion-degree. In our implementation, we find the component-wise RMS values (in RGB space) of color values of all nodes that correspond to a single corrosion value and obtain a one-to-one mapping between corrosion-degree and color. However, we still have corrosion degrees in range $\boldsymbol{[0,1]}$ for which no nodes existed. This results in undefined colors for several corrosion-degrees.

To assign color values to these corrosion-degrees, we use linear interpolation between known colors which are nearest in their corrosion-degree value to the present degree. This completes our one-to-one corrosion-albedo map, which we finally output as a gradient where color at top corresponds to 0% corrosion degree and color at bottom to 100% corrosion degree. In out implementation, the height of the bitmap generated is 1024 pixels (configurable).

4. Feedback

Using the corrosion-albedo map generated in above step, we generate weathering map corresponding to the input image. For each pixel of input image, we calculate the nearest color from the map according to the same distance metric we used earlier. If this nearest color is within *NEIGHBOUR_THRESHOLD*, we assign the pixel corresponding corrosion degree. Else we just mark the pixel red, which depicts that given color was not captured by the control points marked by the user. User can accordingly modify the control points.

Generating texture maps

We now generate following maps to produce final render of a given object mesh and it’s corrosion degree map:

1. Albedo Map
2. Depth Map
3. Metallic Map (Metallicity and Smoothness)
4. Displacement Map

To generate above maps, we require mapping of corrosion degree to corresponding parameters represented in the map. Using the above mentioned technique we generated corrosion-albedo mapping from some sample input image of corroded metallic object. The remaining maps were empirically designed to produce good rendered results (to limit scope of the project).

Algorithm (*mapgen*)

We present following technique for generating above maps corresponding to a given input weathering map:

If the users wishes to increase the resolution of generated maps, we enlarge the given weathering map multiply each pixel's color with a given noise map (fractal Perlin noise with high spatial frequency and average value 0.5 for this project) such that each corrosion-degree changes by a maximum set fraction (***alpha***) to add additional detail.

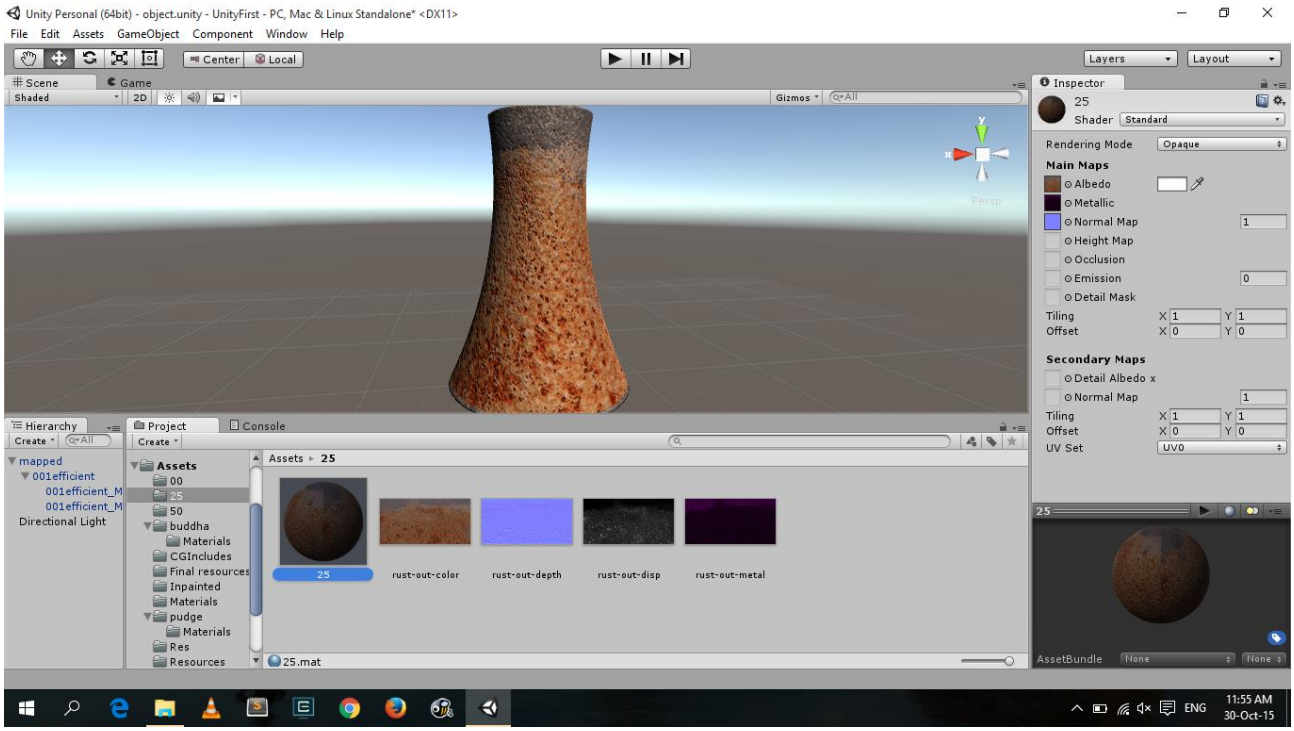
$$degree' = ((1-alpha)+noise*alpha*2)*degree$$

Apart from the weathering map and noise map, we also take appearance manifold in the form of corrosion-albedo map, corrosion-depth map, corrosion-displacement map, corrosion-metallicity map and corrosion-smoothness map. Then a simple mapping is done from *degree'* to albedo, depth, displacement, metallicity and smoothness based on input maps to generate corresponding texture maps.

Rendering

Using the above mentioned technique, we know have texture maps corresponding the parameters which represent appearance of the object. We know take polygon mesh representation of our given input object, and render final results in Unity3D.

As mentioned previously, we use *standard opaque metallic shader* in Unity3D for rendering.



A screenshot of Unity3D 5 with different maps and the mesh

Chapter 3: Experiments & Results

Procedure

In order to compare results, we generated textures from the *mapgen* algorithm using weathering map generated in feedback step of *colorgen* algorithm as input. The albedo texture map generated from *mapgen* algorithm is compared with the original input image. We show the regenerated image and original image in the following results.

Following is the sample input image used in deriving subsequent experimental results:

Original Image

Experiments with Color Space

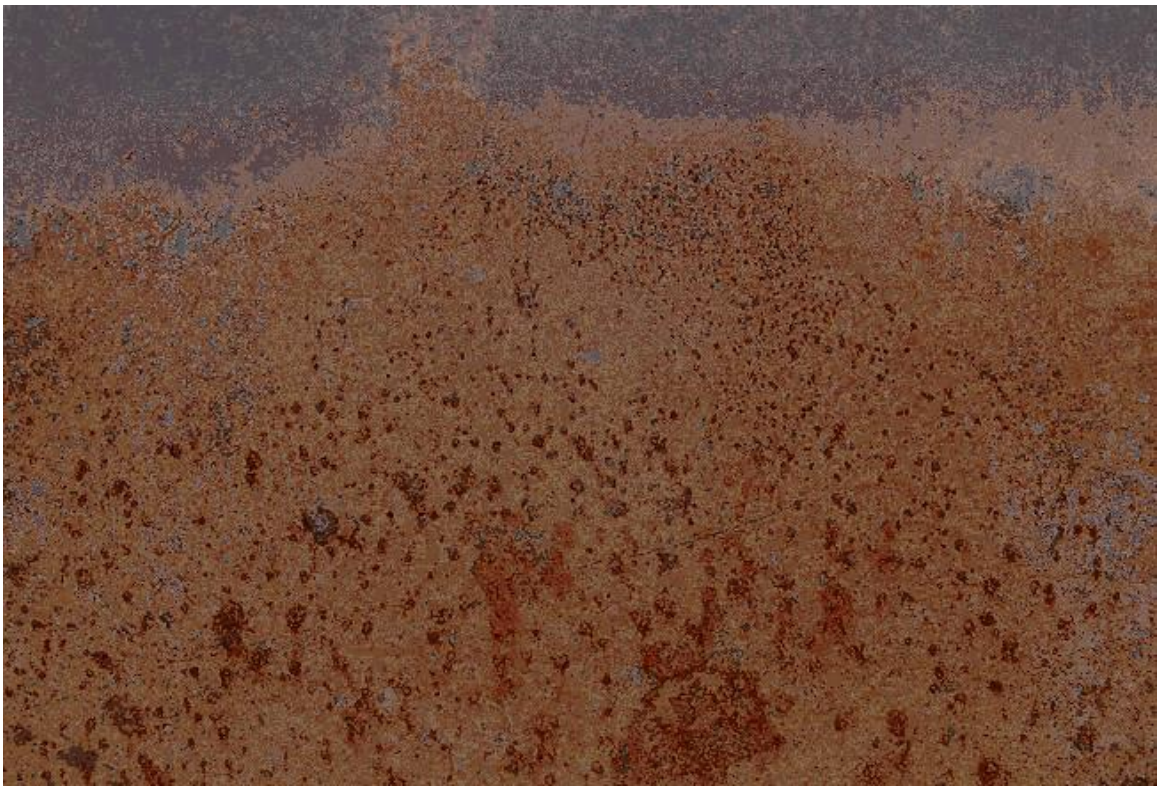
In our algorithm, we experimented with different distance metric (used to create edges in graph) and color space models. The motivation behind different color space model (RGB / YUV / L*ab) was to create a graph in which dissimilarity among different nodes (colors) is in correlation with visual dissimilarity. We used naive Euclidean distance metric and CIE76 distance metric, with later producing more convincing results.

We found best results using Lab color model with CIE76 distance metric as it mimic the nonlinear response of eye. More experiments can be done using advanced distance metric and different color space model to compare and obtain better results.

Following are our experimental results using RGB / YUV / Lab color space models.

RGB Space Euclidean Distance Metric

YUV Space Euclidean Distance Metric

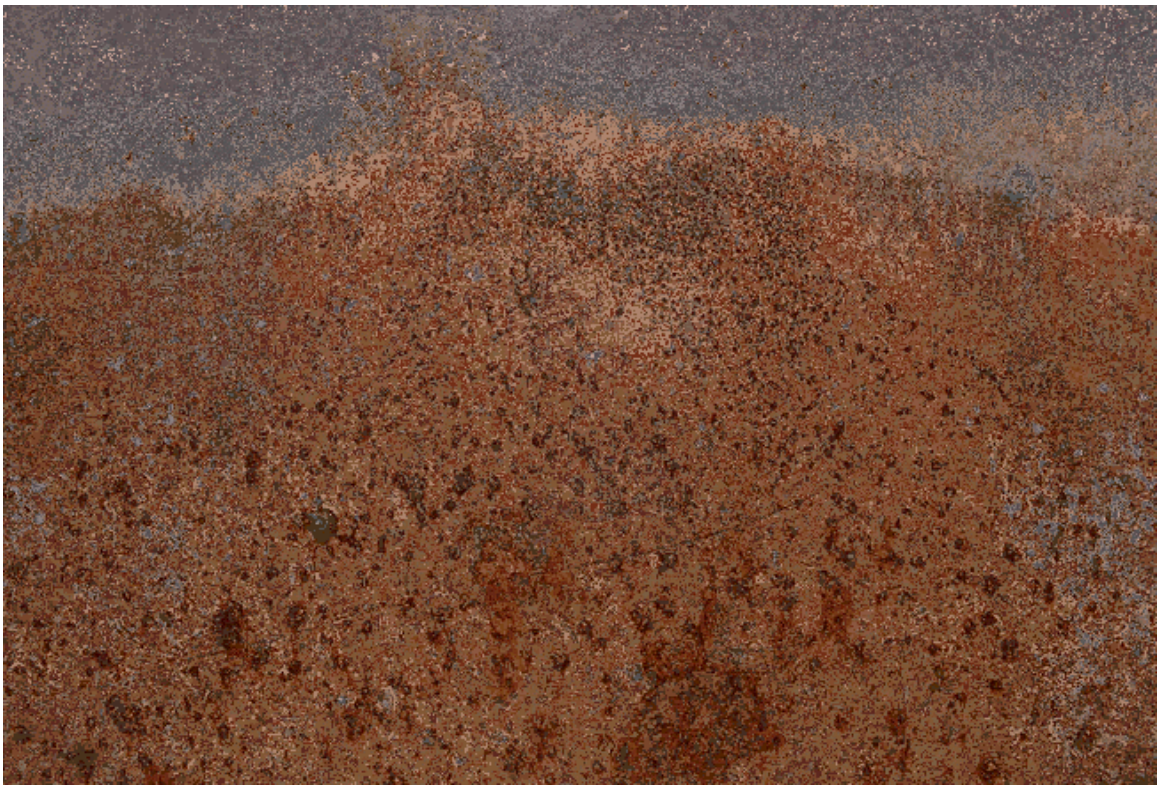


L*ab Space CIE76 ΔE Distance Metric

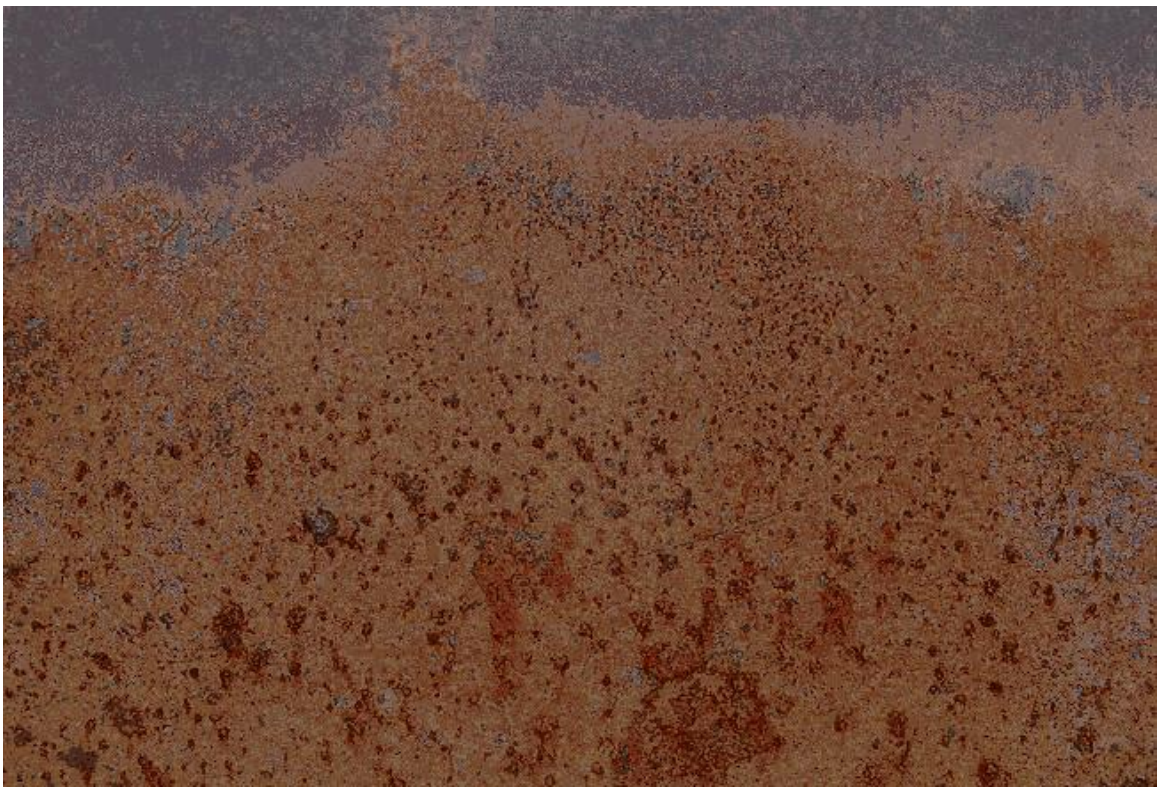
Experiments with MAX_NEIGHBOURS

MAX_NEIGHBOURS is a threshold which is used to limit the degree of each node (color) in the graph. On experimenting with different values of MAX_NEIGHBOURS, we didn't find much difference if value is set above 10. But if the value is too low (3-5), the results were not convincing. The main reason is that for very low value, each node (color) doesn't have sufficient neighbors to capture different colors, thereby resulting in in sparse graph and more disconnected components. When this value increases, number of different colors among neighborhoods increases making graph more useful for computation of corrosion values.
We also noted that with higher value, graph becomes more dense and subsequent computation over it become too expensive. We found the value of 12 as the optimal one, as it makes graph sufficiently connected and also doesn't make computation too expensive.

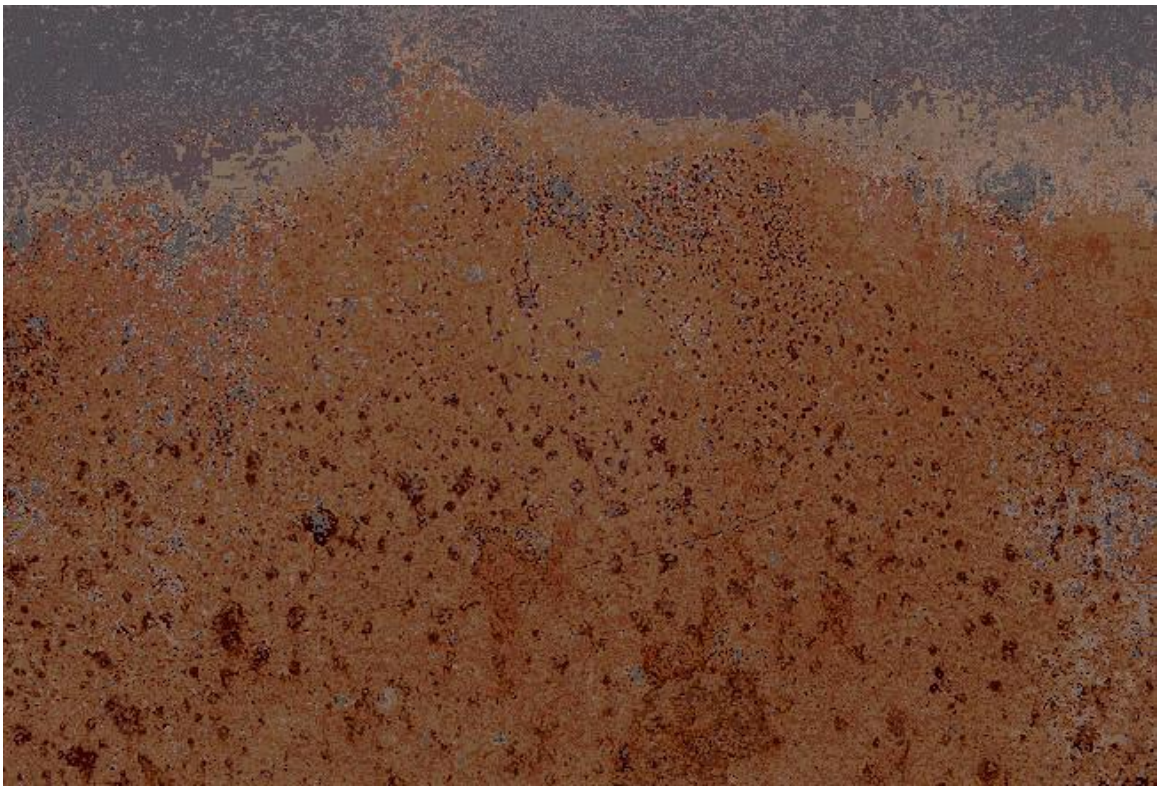
Following are our results with value of MAX_NEIGHBOURS = 3, 12, and 18:



MAX_NEIGHBOUR=3, Notice excessive noise



MAX_NEIGHBOUR=12, Very similar to original

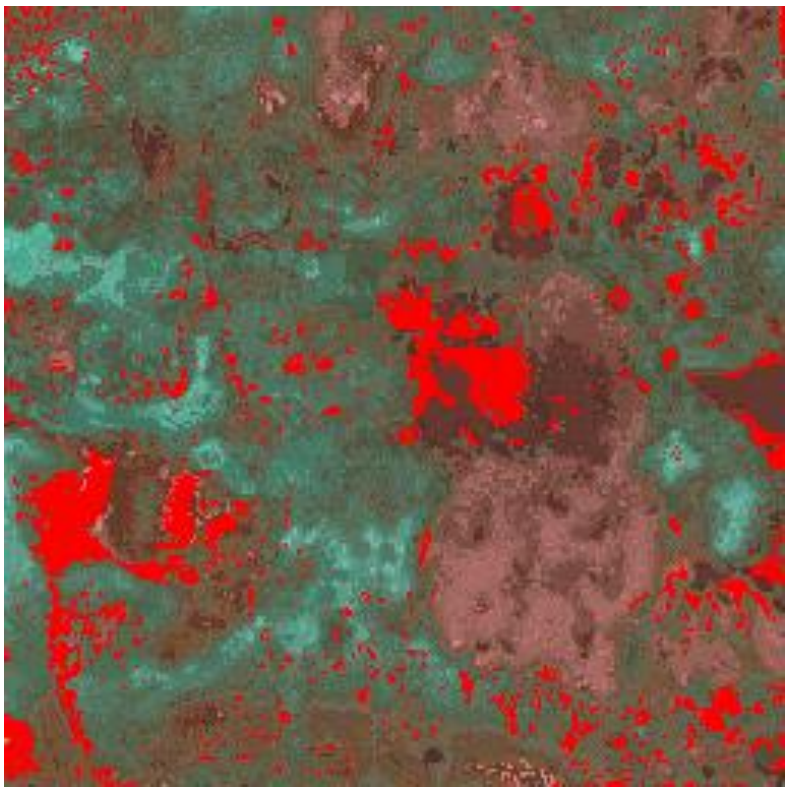


MAX_NEIGHBOUR=18, Almost identical to previous case

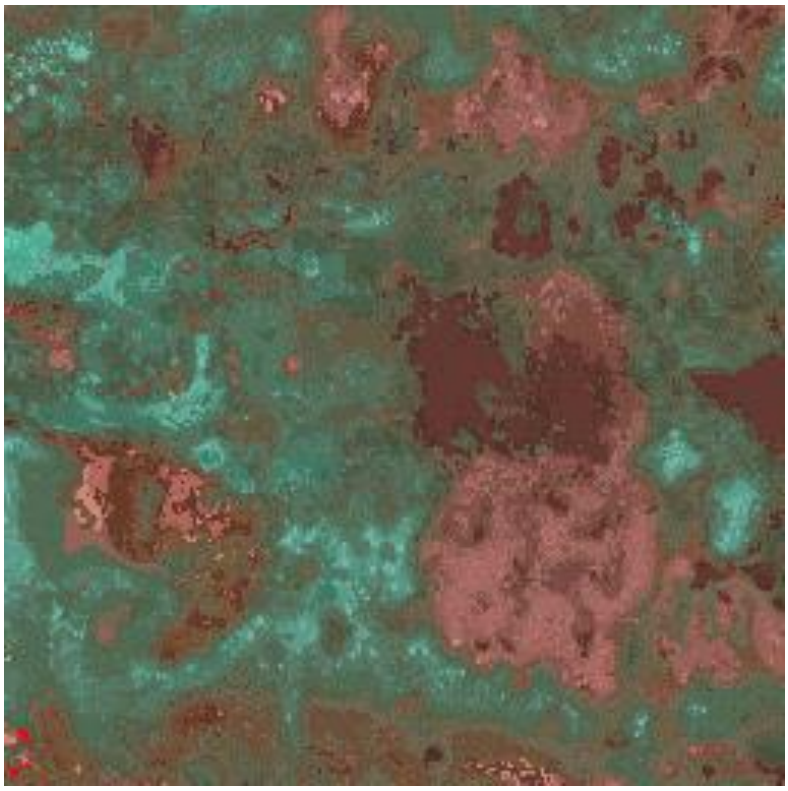
Experiments with Control Points

In our algorithm, we took control points as input from the user. As mentioned previously, higher are the control points better are the results because it gives us good initial idea about relation between corrosion degree and color. So incase user makes mistakes in entering control points we generate weathering map (in feedback step of *colorgen*) which gives an idea about how colors are captured. Using the weathering maps, user can ascertain whether necessary colors are indeed captured. Otherwise he can change the control points to capture that color.
Following are our results using different control points:

Original Image



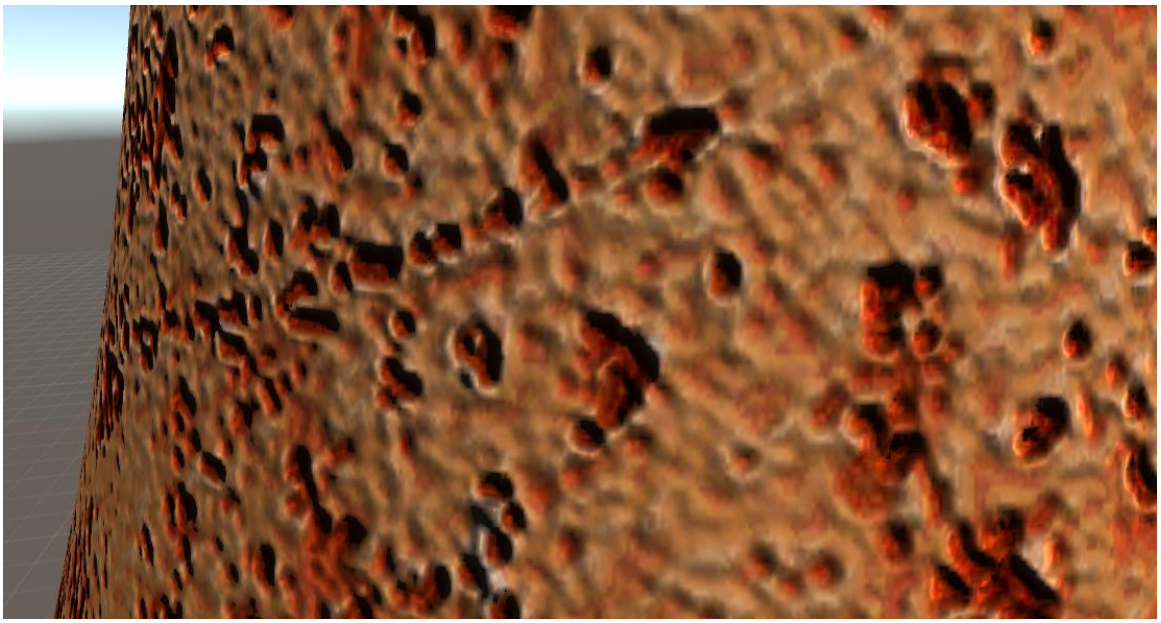
4 control points, some parts not captured



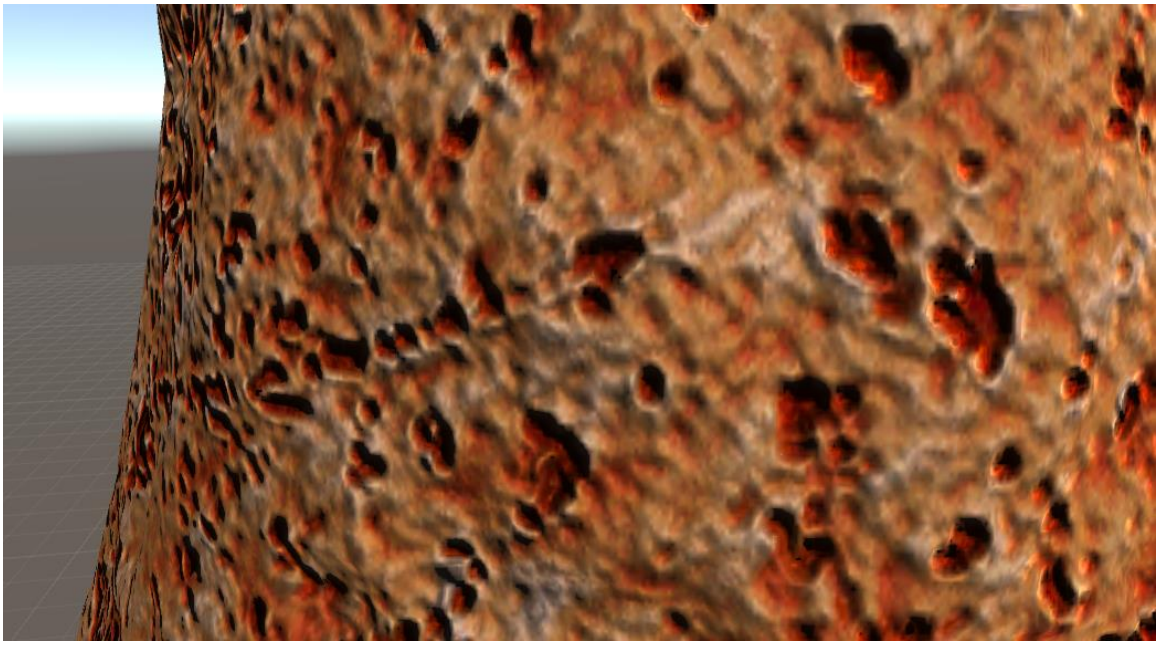
6 control points, almost all areas captured

Experiments with Perlin Noise

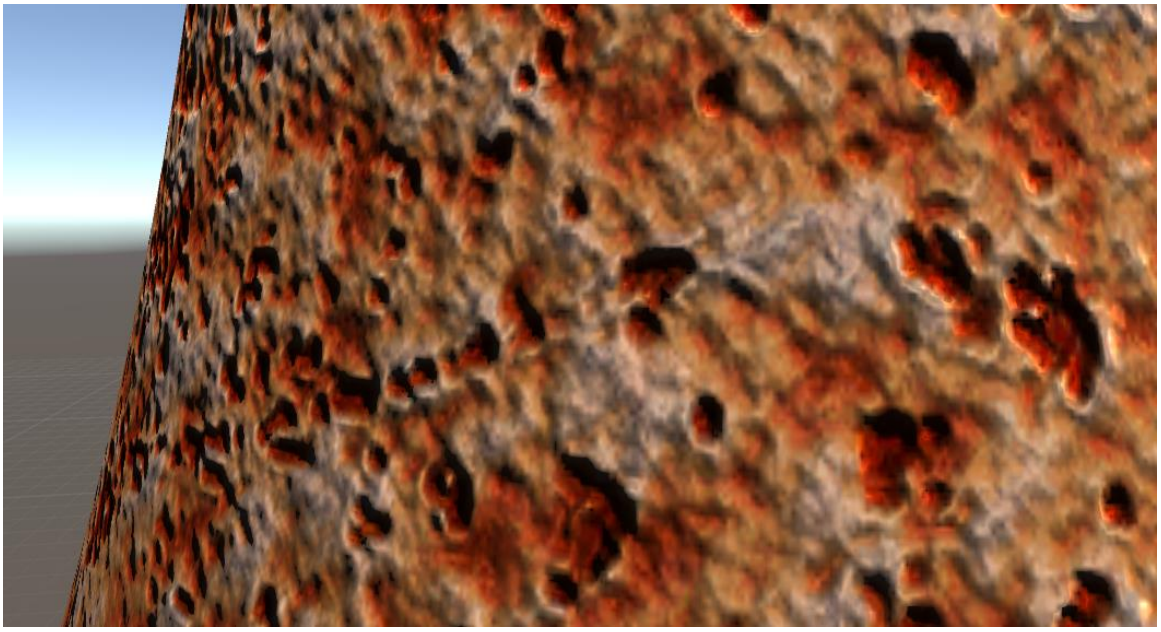
We used ***alpha*** parameter in *mapgen* algorithm to alter the corrosion degree and adding more details to the base weathering degree map. Experimentally, we observed that for very low alpha, there are very less details in weathering map thereby appearing blurred when camera is near the surface. Increase alpha adds more details, however, very high alpha leads to visual dissimilarity from original even at far camera positions. Optimal results are obtained for alpha around *0.15* to *0.25*.



Final Render, Alpha=0.0, Notice lack of details



Alpha=0.25, Some details are added



Alpha=0.5, Notice excessive manipulation of details

Final results

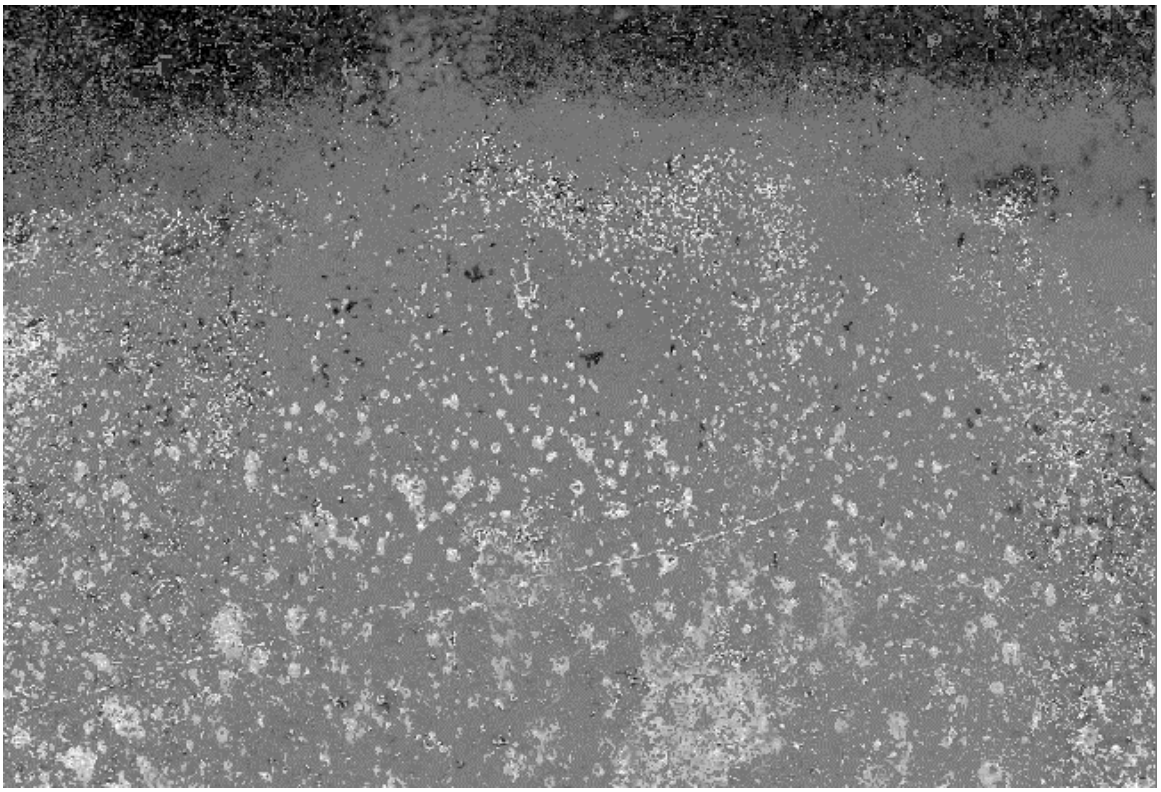
We now compare our results with results from an implementation of original appearance manifold method discussed by *Wang et. al [1]*, a spatial coloring modification to the appearance manifold method by *Nisha Jain* (and its YUV color space variant).

Sample #1

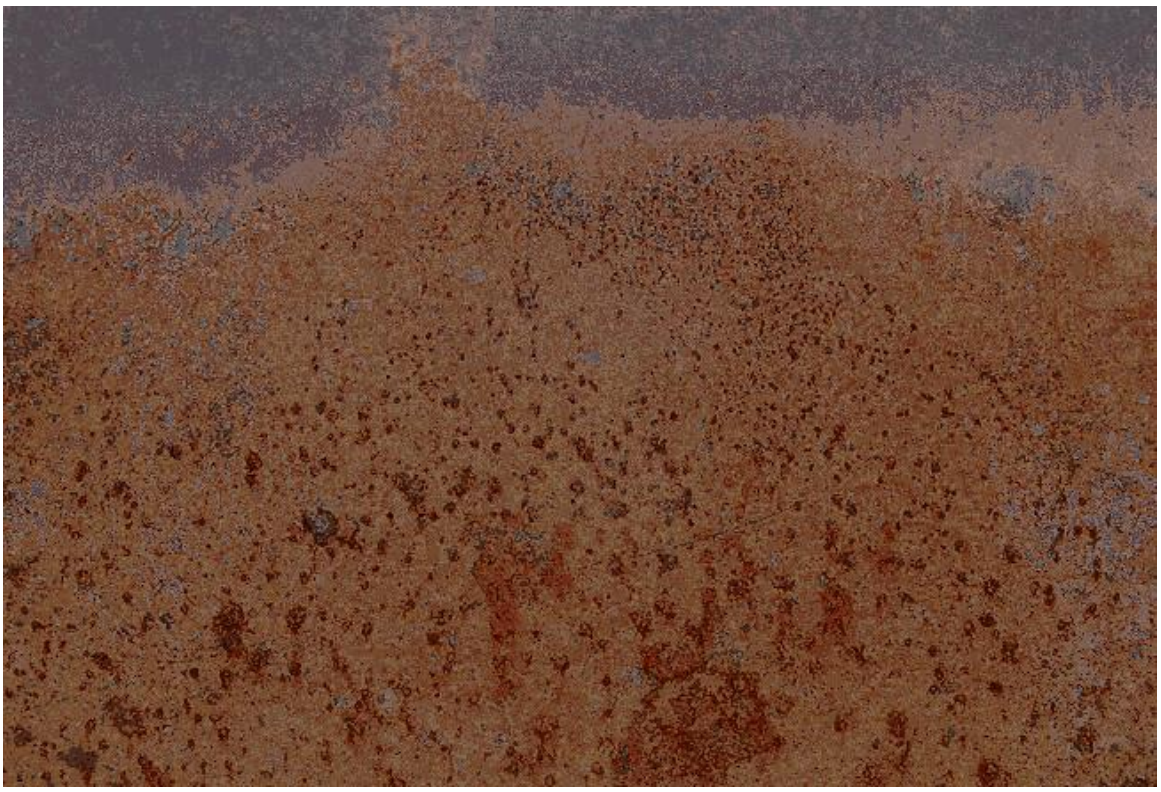
It can be seen that appearance manifold method outputs a washed out image with loss of visual difference b/w paint color and rust color (which is due to it using only 2 sets of points: least and most corroded). Our method performs better at producing more visually similar image to the original image. It captures more number of colors and various features in the image preserved better as compared to the appearance manifold method.

Original Input Image, Iron Surface

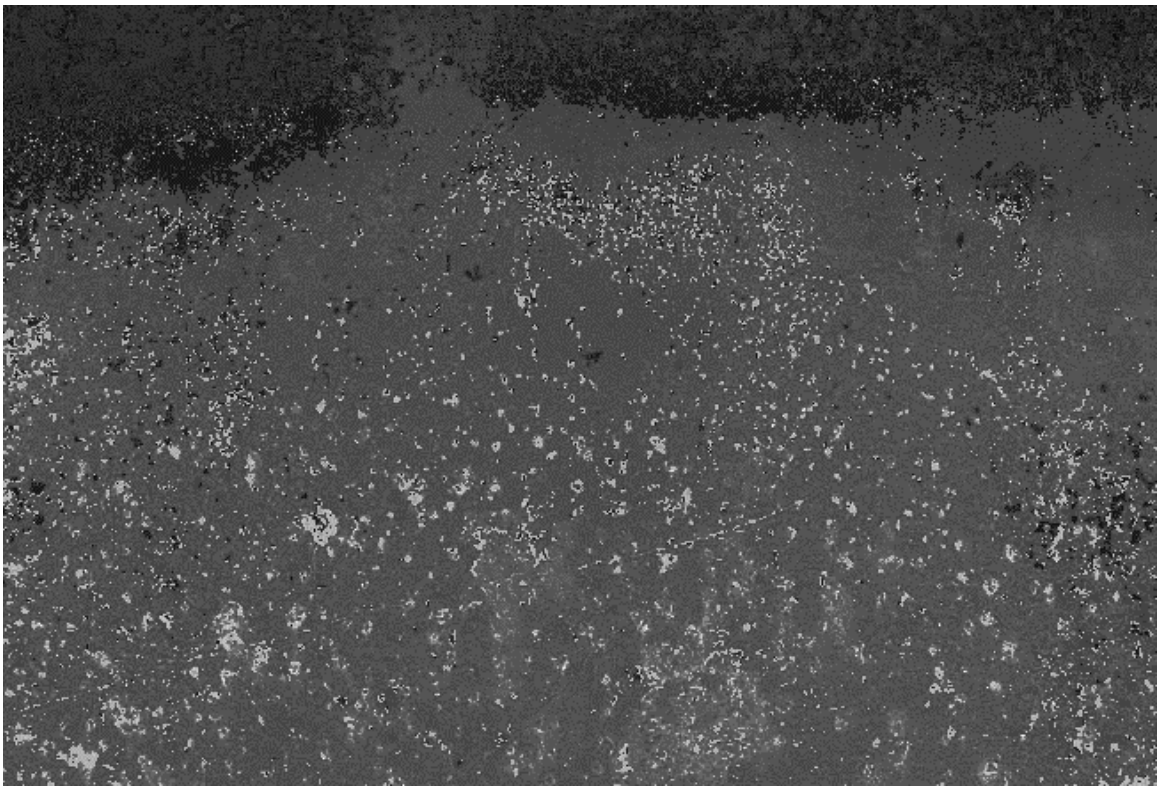
Albedo map output, Original Appearance Manifold Method



Weathering map output, Original Appearance Manifold Method



Albedo map output, Our Method



Weathering map output, Our Method

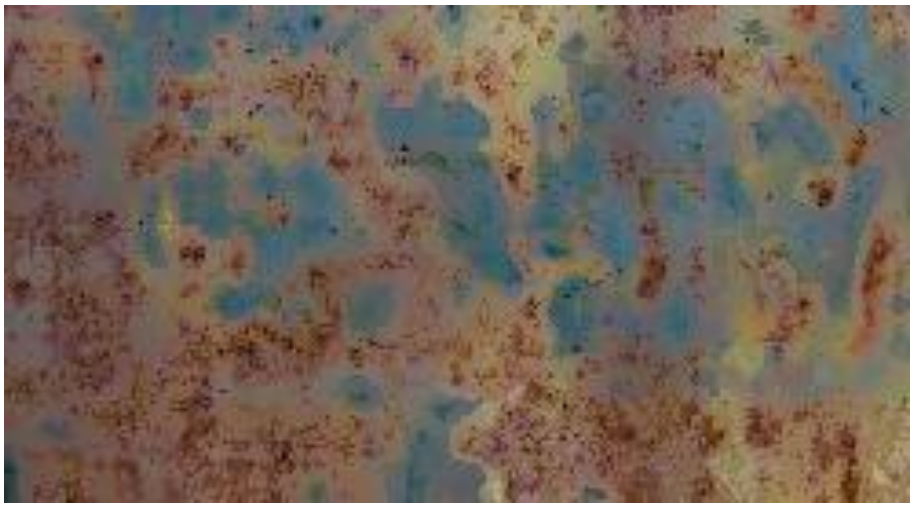
Output Gradient

Final Render, Our Method

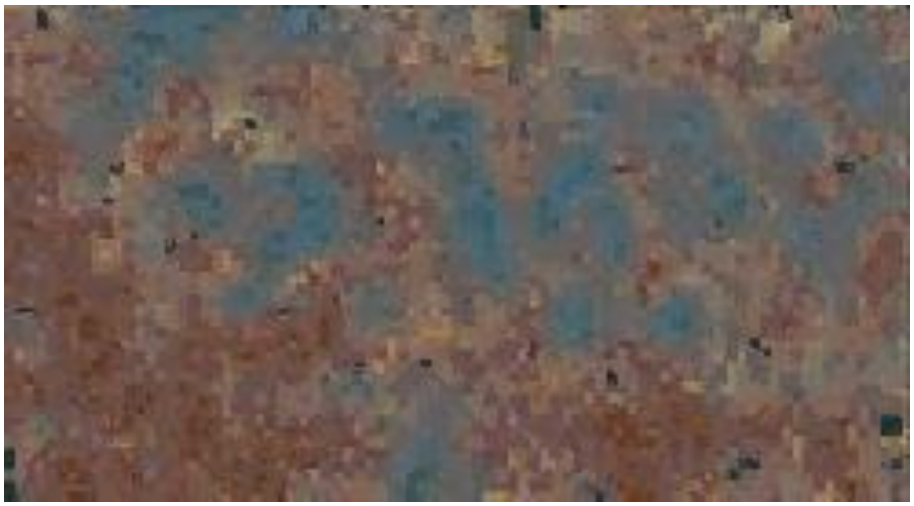
Sample #2

In this sample,

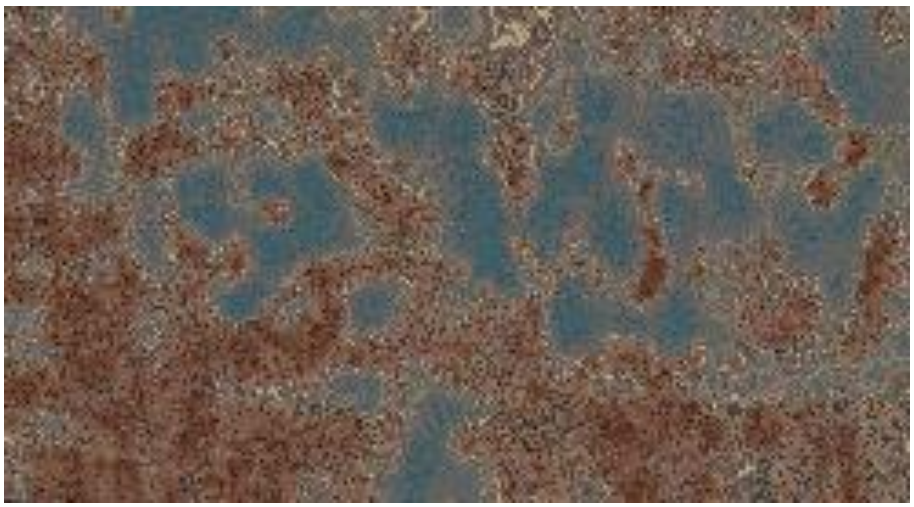
- Appearance manifold method results in a very blurred and feature-less image
- Point-coloring maintains features, however adds too much noise (both in RGB and YUV space)
- Our method performs well in both detail and color aspect, visually very similar to the original image.



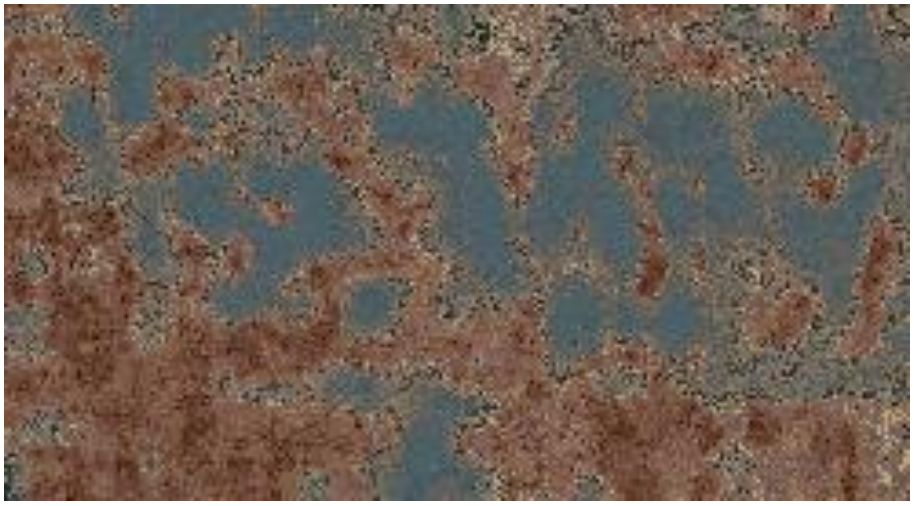
Original Image, Painted Iron Surface



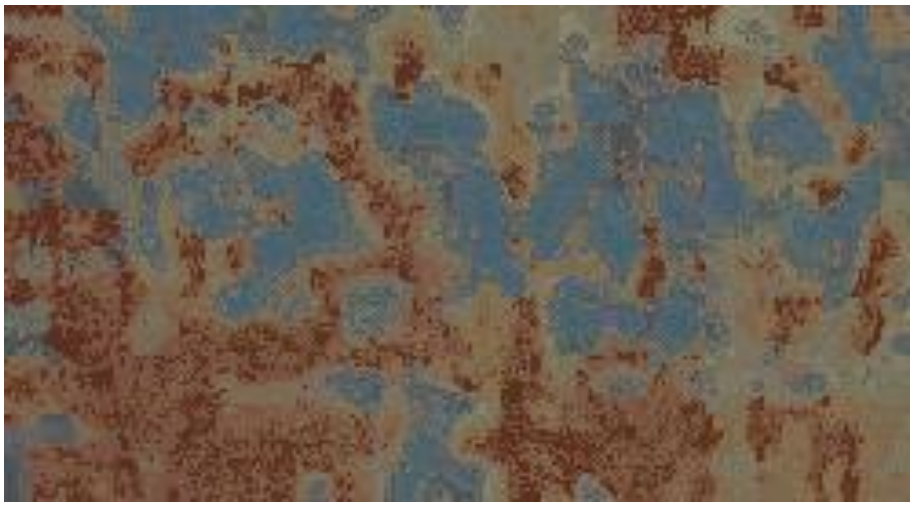
Albedo Map, Original Appearance Manifold



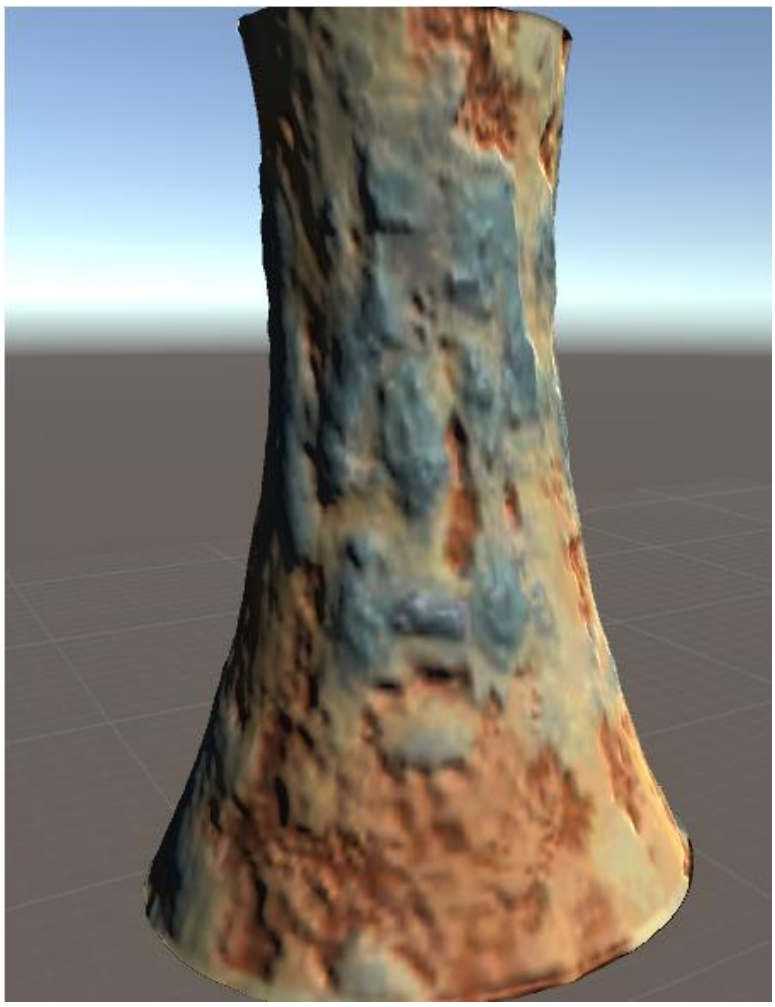
Albedo Map, Appearance Manifold with Point Coloring (RGB)



Albedo Map, Appearance Manifold with Point Coloring (YUV)



Albedo Map, Our Method



Final Render, Our Method

Conclusion

Our algorithm performs better than the original appearance manifold method (*Wand et. al*), and the variations made using point-coloring methods (*Nisha Jain*), as evident from the results. It performs significantly better when there are different hues present in the corrosion stages of the sample image (Painted surfaces for example) due to ability of user to specify control points across the full corrosion degree range.

Improvements

Several improvements can be made to our method. Here are some areas we felt can significantly improve the results:

- Instead of using arithmetic mean while combining various colors for a single corrosion degree, using other central tendencies (Geometric median)
- Space partitioning algorithms can be used at several places to improve efficiency of the algorithm so that we don't need to down sample the colors.
- Consider spatial location of nodes into the graph while leaking corrosion degree in the whole graph.
- Generate depth, metallicity and smoothness maps also using empirical or physically based models, using surface images under different lighting conditions and depth scans as inputs.

References

1. http://doi.acm.org/10.1145/1179352.1141951

2. http://web.eecs.utk.edu/~huangj/papers/polygon.pdf

3. http://etd.uwaterloo.ca/etd/rjaroszkiewicz2003.pdf