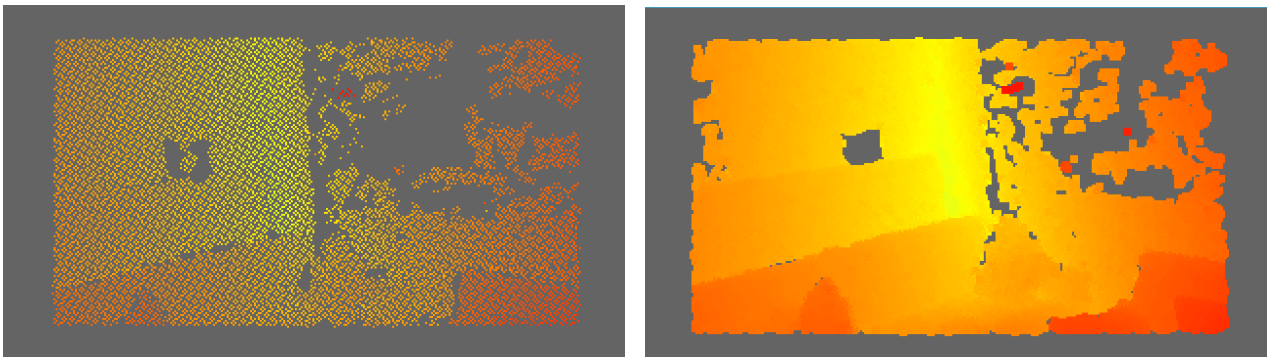
Working with Yellowstone depth images

1. Applying a nearest-neighbor filter

The depth images on Yellowstone are created by taking a point cloud and projecting it onto an image plane. Since the points in the cloud are somewhat sparse, the depth images tend to have a lot of gaps. Here we present a simple nearest-neighbor based filter to fill holes. The filter will not interpolate values, which preserves depth image edges.



Here is an example of the original scene without filtering, and with a filtering with a window parameter of 2. The window size parameter refers to the number of neighbors visited around a given pixel. Thus, a value of 1 will result in 3x3 pixel patches, 2 will result in 5x5 pixel patches, etc.

Pseudocode:

```
set all pixels in output_image to 0
for all pixels input_image:
  if input_z is not 0, set output_z to input_z
  else
    for all pixels in neighbor window of size w around current pixel
      if neighbor_z iz not 0
        if output_z is 0, set output_z to neighbor_z
        else set output_z to the smaller of (output_z, neighbor_z)
    end for
end for
```

C++ code:

```
void FilterDepthImage(const engine_data::DepthImage& depth_image_in,
                      int filtering_window,
                      engine_data::DepthImage* depth_image_out) {
  depth_image_out->SetZero();
  const int rows = depth_image_in.rows();
  const int cols = depth_image_in.cols();

  // Go over all pixel coordinates.
  for (int v = 0; v < rows; ++v) {
    for (int u = 0; u < cols; ++u) {
      uint16_t& z_out = depth_image_out->AtMutable(v, u);
      const uint16_t& z_in = depth_image_in.At(v, u);
      if (z_in != 0) {
        z_out = z_in;
      } else {
        // Iterate over local window and perform z-buffering.
        const int v_start = std::max(v - filtering_window, 0);
        const int u_start = std::max(u - filtering_window, 0);
        const int v_end = std::min(v + filtering_window, rows - 1);
        const int u_end = std::min(u + filtering_window, cols - 1);

        for (int vv = v_start; vv < v_end; ++vv) {
          for (int uu = u_start; uu < u_end; ++uu) {
            const uint16_t& z_neighbor = depth_image_in.At(vv, uu);
            if (z_neighbor != 0) {
              if (z_out == 0)
                z_out = z_neighbor;
              else
                z_out = std::min(z_out, z_neighbor);
            }
          }
        }
      }
    }
  }
}
```