Optimizing the Shape Completion Pipeline

David Watkins Columbia University
New York, New York USA
May 11, 2017
djw2146@columbia.edu

ABSTRACT

Expanding on the work done by Jake Varley et al. for the Shape Completion Enabled Robotic Grasping[3], I performed a series of optimizations that would enhance the pipeline to increase its performance and its flexibility. The marching cubes algorithm has been rewritten to support GPU operations, preliminary code has been written for completing entire scenes based on work done by Evan Shelhamer et al.[2], and written a headless depth renderer to help generate scenes for training data much faster than the current pipeline. These three contributions will prove to effectively push forward the shape completion project to a much more usable state for not only our lab but also any labs that may choose to use this software in the future.

1. INTRODUCTION

In this section I will go over the various technologies and platforms that were used to build the enhancements to the shape completion pipeline as well as describe the shape completion pipeline itself.

1.1 CUDA

CUDA is an extension of C++ whereby specific instructions are added to allow for GPU programming. This allows a programmer to develop their program to support several orders of magnitude higher parallelization than what is possible on a CPU. Modern day GPUs have evolved into highly parallel multi-core systems that allow for efficient manipulation of large blocks of data. There are many different versions of the CUDA software development kit (SDK) but for the purposes of this paper we have used primarily 3.5.

In order to process elements on the GPU, a developer must first transfer the data that he wants processed onto the GPU itself, then call a compiled kernel to then run on the GPU itself with references to the data loaded onto the GPU, then transfer the results back. This might look like the following:

```
texture<float, 2, cudaReadModeElementType> tex;

void foo()
{
        cudaArray* cu_array;

        // Allocate array
        cudaChannelFormatDesc description =
                cudaCreateChannelDesc<float>();
        cudaMallocArray(&cu_array,
                        &description,
                        width,
                        height);

        // Copy image data to array
        cudaMemcpyToArray(cu_array,
                      image,
                      width*height*sizeof(float),
                      cudaMemcpyHostToDevice);

        // Set texture parameters (default)
        tex.addressMode[0] = cudaAddressModeClamp;
        tex.addressMode[1] = cudaAddressModeClamp;
        tex.filterMode = cudaFilterModePoint;
        tex.normalized = false;

        // Bind the array to the texture
        cudaBindTextureToArray(tex, cu_array);

        // Run kernel
        dim3 blockDim(16, 16, 1);
        dim3 gridDim((width + blockDim.x - 1)/ blockDim.x,
                                     (height + blockDim.y - 1)
                                     1);
        kernel<<< gridDim, blockDim, 0 >>>
             (d_data, height, width);

        // Unbind the array from the texture
        cudaUnbindTexture(tex);
} //end foo()

__global__ void kernel(float* odata, int height, int width)
{
        unsigned int x =
                 blockIdx.x*blockDim.x + threadIdx.x;
        unsigned int y =
                 blockIdx.y*blockDim.y + threadIdx.y;
        if (x < width && y < height) {
                float c = tex2D(tex, x, y);
                odata[y*width+x] = c;
        }
}
```

This program is loading a image texture onto the GPU and for each pixel loads it into an array on the gpu. This trivial example starts with a call *cudaMemcpyToArray* which takes the host memory and transfers it over to the GPU. Typically these calls can be expensive the more data is being transferred, therefore it is generally advised to schedule these transfers while data is being simultaneously processed via a kernel on the GPU. Modern GPUs support simultaneous memory operations as well as processing operations which can help amortize the cost of transferring data. Later in this paper I will discuss how I utilized this amortization to reduce the overall cost of computing the marching cubes on a completed mesh in order to get an order of magnitude of performance boost.

The reason why a GPU is effective for this particular project is because the original processing of individual voxels had a triple-nested for loop for each dimension (x, y, z) where each iteration of the loop is independent of another. When programs have such loops it is better to spawn a series of threads to execute each of those iterations in parallel on a GPU where there are thousands of cuda cores which can execute the code at the same time. The actual implementation details of how this was done will be discussed later.

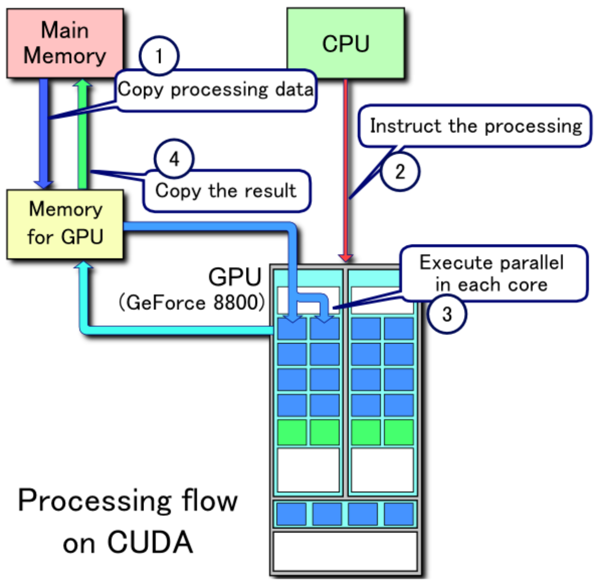

Figure 1: The pipeline of a GeForce 8800 GPU describing the process of 1) copying data from the CPU to the GPU 2) initiating a kernel call 3) executing a copy of the kernel across the thousands of threads within a GPU and finally 4) copying the result back to the CPU.

1.2 OpenGL

OpenGL is a cross-language API for rendering 2D and 3D vector graphics. It is designed to interact with a GPU to allow for hardware accelerated rendering of various shaded modules. This pipeline is very useful for computing things such as depth field maps and rendering of 2D images of a 3D scene very quickly especially when compared to something more complex such as a raytracing renderer. OpenGL in particular uses was is called a pipeline renderer whereby it uses a series of matrix transformations of the triangles of a mesh to compute what the resultant 2D image will look like as well as compute the distance from the camera origin to the object itself. For the purposes of this project OpenGL was used as a simple way of generating depth renderings of a mesh to determine per-pixel distances to each element in the mesh and thereby generate a voxel map of an object. More about why this was useful in a later section.

1.3 Shape Completion Pipeline

The shape completion pipeline was a robotic grasp planner that used a partial view of a mesh and a train neural net to generate a completed object even if certain regions of the scene were occluded. It uses a convolutional neural network (CNN) to do this where a pointcloud captured from a single point of view is fed into the CNN which fills the occluded regions of the scene and then a grasp planner takes the completed object and plans grasps on it. After the CNN returns an approximated voxel grid. This voxel grid is merged with the original pointcloud captured from a 3d depth camera using the marching cubes algorithm. The marching cubes algorithm has multiple steps where performance can be enhanced, such as the quadratic programming and depth fields stages which are highly parallelizable. The major contribution here is increasing the performance of this pipeline to reduce the time spent computing the valid object model.[3]

1.4 CNNs and Scene Completion

Again, using convolutional neural networks, a team at UC Berkeley trained a model to identify objects within a RGBD image through per-pixel labeling of objects of 40 known categories. These labels are useful as they can be used within a robotics pipeline to segment an image very rapidly and then pull specific sections of the RGBD image that are of interest - such as getting a partial mesh to be used within the shape completion pipeline. The data for this system was preprocessed and in the case of the NYU depth database[1] were touched up by a human editing the scene. Through our testing we showed that we did not need to touch up the scene and I will discuss this in more detail later.[2]

2. TIMELINE

For this project the general timeline followed this trajectory:

- September 6th - October 7th: Working on optimizing the shape completion pipeline by implementing existing functionality using the CUDA SDK instead.
- October 7th - November 20th: Investigating the fully convolutional neural network system that Trevor Darrell's lab put out[2] and determine how to integrate it with our current shape completion pipeline to create a scene completion system.
- November 1st - December 15th: Build a depth renderer that takes in a mesh from a kinect and gives a per-pixel depth image to be used for voxelization to replace the current gazebo framework

A more meticulous schedule was not maintained and therefore this is the highest granularity I was able to muster for the purposes of this report.

3. PROCESS

In this section I discuss the implementation details of each of the different projects I worked on this semester including specific coding examples and files appended to the end of this document describing what was done.

3.1 Shape Completion Optimization

The shape completion pipeline was relatively easy to add to considering it was a finished application and it was a simple optimization task to make it run faster using CUDA programming. I modified two files: quadprog.cpp and dfields.cpp. Both of these files had nested for loops which took a significant amount of CPU time to compute where each loop was independent of any other iteration of the loop.

3.1.1 Distance Fields

Take the following code that was the innermost loop of the fastPerim function within dfields.cpp, where i,j,k are the dimensions of the voxel grid:

```
...
(*g)[i][j][k]=1.0;
int neg_i=1;int neg_j=1;int neg_k=1;
int pos_i=1; int pos_j=1; int pos_k=1;
if(i==0)neg_i=0;
else if(i==g->dims[0]-1)pos_i=0;
if(j==0)neg_j=0;
else if(j==g->dims[1]-1)pos_j=0;
if(k==0)neg_k=0;
else if(k==g->dims[2]-1)pos_k=0;
//search neighboring voxels for different value
for(int i_=i-neg_i; i_<=i+pos_i; i_++){
        for(int j_=j-neg_j; j_<=j+pos_j; j_++){
                for(int k_=k-neg_k; k_<=k+pos_k; k_++){
                        if((*volume_grid)[i][j][k] !=
                        (*volume_grid)[i_][j_][k_]){
                                (*g)[i][j][k]=0.0;
                        }
                }
        }
}
...
```

This entire code block can be extracted into its own function within the GPU where it iterates through each voxel within the voxel grid and calculates the perimeter for each element. This is done by checking the center voxel of a 3x3x3 voxel grid within the larger voxel grid and determining if there is a voxel presence around the current voxel. Normally computing these can be slower because global memory accesses, which the volume_grid would be in the global memory scope, are much slower. However in more modern versions of CUDA these operations have been optimized and they are fast enough for relatively small grids - and for a 40x40x40 grid with only 32000 elements it would only have a 64KB footprint which is very small by GPU standards. The voxel grid would have to be several megabytes in size before performance drops would be seen.

There are several other functions within the distance fields file which were able to be optimized. The getsqdist function had three triple-nested for loops in a row each of which had an internal block of code in which the previous iteration had no impact on a future iteration. I took each of these three loops and turned them into three CUDA kernel functions each of which operated on a volume_grid which was simply a 3d voxel grid.

```
__global__ void sqdistpt1(float* dist_grid, float*
↪  dest_grid, int xdim, int ydim, int zdim) {
        int center =
        ↪  blockIdx.x*blockDim.x+threadIdx.x;

        if(center < xdim * ydim * zdim) {

                int i,j,k,var=center;
                get3DIndex(i,j,k,var,xdim,ydim,zdim);

                //get squared dist to closest 0 in
                ↪  row (i's)
                float min = 100000;
                for(int index=0; index<xdim;
                ↪  index++){
                        float dist = 100000;
                        if(dist_grid[getFlatIndex(index,
                        ↪  j, k, xdim, ydim,
                        ↪  zdim)]==0.0){
                                dist = (index-
                                ↪  i)*(index-i);
                        }
                        if(dist<min){
                                min=dist;
                        }
                }
                //set value of voxel to min q dist
                dest_grid[center]=(float)min;
        }
}

__global__ void sqdistpt2(float* dist_grid, float*
↪  dest_grid, int xdim, int ydim, int zdim) {
        int center =
        ↪  blockIdx.x*blockDim.x+threadIdx.x;

        if(center < xdim * ydim * zdim) {

                int i,j,k,var=center;
                get3DIndex(i,j,k,var,xdim,ydim,zdim);

                //get squared dist to closest 0 in
                ↪  row (i's)
                int min = 100000;
                for(int index=0; index<ydim;
                ↪  index++){
                        int dist =
                        ↪  dist_grid[getFlatIndex(i,
                        ↪  index, k, xdim, ydim,
                        ↪  zdim)] + (index - j) *
                        ↪  (index - j);
                        if(dist<min) min=dist;
                }
                //set value of voxel to min q dist
                dest_grid[center]=(float)min;
        }
}

__global__ void sqdistpt3(float* dist_grid, float*
↪  dest_grid, int xdim, int ydim, int zdim) {
        int center =
        ↪  blockIdx.x*blockDim.x+threadIdx.x;

        if(center < xdim * ydim * zdim) {

                int i,j,k,var=center;
                get3DIndex(i,j,k,var,xdim,ydim,zdim);

                //get squared dist to closest 0 in
                ↪  row (i's)
                int min = 100000;
                for(int index=0; index<zdim;
                ↪  index++){
                        int dist =
                        ↪  dist_grid[getFlatIndex(i,
                        ↪  j, index, xdim, ydim,
                        ↪  zdim)] + (index - k) *
                        ↪  (index - k);
                        if(dist<min) min=dist;
                }
                //set value of voxel to min q dist
                dest_grid[center]=(float)min;
        }
}
```

By separating out these functions I could remove the need for several triple nested for loops calls. Conveniently the getsqdist function took as an argument the result of the fastPerim function, so I was able to use the data as it existed on the GPU for future processing as well. This saved time that would otherwise be spent copying the result back to the CPU and back to the GPU. Once the three calls to the separated getsqdist functions were called I was able to write a sqrt function that called sqrt on each voxel in the grid to get the true distance of each voxel grid from the neighbors.

```
__global__ void sqrtgpu(float* dist_grid, float*
↪  dest_grid, int xdim, int ydim, int zdim) {
        int center =
        ↪  blockIdx.x*blockDim.x+threadIdx.x;

        if(center < xdim * ydim * zdim) {
                dest_grid[center] =
                ↪  sqrt(dist_grid[center]);
        }
}
```

Putting all of these functions together I was able to define a series of kernel calls each of which were able to use the result from the previous CUDA kernel call.

```
gridPtr dfield_gpu(gridPtr volume_grid) {
        float* flat_g = flatten(volume_grid);
        int xdim = volume_grid->dims[0], ydim =
        ↪  volume_grid->dims[1], zdim =
        ↪  volume_grid->dims[2];
        int g_size = xdim * ydim * zdim;

        float* dest_grid, *src_grid;
        cout << "Initializing dfields arrays" <<
        ↪  endl;
        GPU_CHECKERROR(cudaMalloc((void**)
        ↪  &dest_grid, g_size * sizeof(float)));
        GPU_CHECKERROR(cudaMalloc((void**)
        ↪  &src_grid, g_size * sizeof(float)));
        GPU_CHECKERROR(cudaMemcpy(src_grid,
        ↪   flat_g, g_size * sizeof(float),
        ↪   cudaMemcpyHostToDevice));

        int BLOCK_SIZE = 256;
        int NUM_BLOCKS =
        ↪   (int)ceil(double(g_size)/BLOCK_SIZE);

        //Alternate buffers so we don't have to
        ↪   move data around on the device
        //Buffers effectively alternate between
        ↪   input and output
        cout << "dfield Fast perim gpu" << endl;
        fastPerimGPU<<<NUM_BLOCKS,
        ↪   BLOCK_SIZE>>>(src_grid, dest_grid,
        ↪   xdim, ydim, zdim);

        cout << "dfield sqdist" << endl;
        sqdistpt1<<<NUM_BLOCKS,
        ↪   BLOCK_SIZE>>>(dest_grid, src_grid,
        ↪   xdim, ydim, zdim);
        sqdistpt2<<<NUM_BLOCKS,
        ↪   BLOCK_SIZE>>>(src_grid, dest_grid,
        ↪   xdim, ydim, zdim);
        sqdistpt3<<<NUM_BLOCKS,
        ↪   BLOCK_SIZE>>>(dest_grid, src_grid,
        ↪   xdim, ydim, zdim);

        cout << "dfield sqrt" << endl;
        sqrtgpu<<<NUM_BLOCKS,
        ↪   BLOCK_SIZE>>>(src_grid, dest_grid,
        ↪   xdim, ydim, zdim);

        //Copy data back to device
        GPU_CHECKERROR(cudaMemcpy(flat_g,
        ↪   dest_grid, g_size * sizeof(float),
        ↪   cudaMemcpyDeviceToHost));

        cudaThreadSynchronize();

        cudaFree(dest_grid);
        cudaFree(src_grid);

        cudaThreadSynchronize();

        return unflatten(flat_g, volume_grid,
        ↪   xdim, ydim, zdim);
}
```

By alternating the passing of src_grid and dest_grid I was able to reuse the results from previous kernel calls. I determined the number of blocks needed for each kernel call by taking the number of voxels in the grid and dividing by the number of threads per block where each thread was assigned to a single voxel. This held true for every single function call within dfields.cpp. For the purposes of good code design - if CUDA was present on the system compiling the shape completion pipeline it would compile the dfields.cu file, and if there was no CUDA present it would compile the original dfields.cpp. I did not change the syntax of the dfield function which meant I could add compiler flags to designate which one to use within the header.

3.1.2 Quadratic Programming

The quadratic programming algorithm is a much trickier algorithm to implement as it requires a series of discrete step calls in order to perform the necessary optimization of the voxel grid. The step function is run approximately 512 times to get a reasonable result for the voxel grid. Originally within each of these steps was a single for loop where each iteration was not dependent on another. Because of this it made it a perfect candidate for GPU optimization - however it was tricky because each step is dependent on the result of the previous one. Therefore I determined I was able to flip the input and output arrays on the doStep function on each iteration thereby utilizing the data from the previous iteration and providing a new result without reading memory back to the CPU or copying memory over on the GPU.

This transformation was from this version of the code:

```
...
for(int i = 0; i < args->iter; i++){
        if(i % 2){
                in = buf2;
                out = buf1;
        }
        else{
                in = buf1;
                out = buf2;
        }

        for(int r = 0; r < size; r++){
                doStep(r, *in, *out, ir, jc, pr,
                ↪   args->invdg, args->lb,
                ↪   args->ub);
        }
}
...
```

to the following code:

```
...
for(int i = 0; i < args->iter; i++){
           if(i > 0) {
                   float* swap = inCu;
                   inCu = outCu;
                   outCu = swap;
           }
    doStepDevice<<<NBLOCKS,BLOCK_SIZE>>>(inCu,
    ↪   outCu, irCu, jcCu, prCu, invdgCu, lbCu,
    ↪   ubCu, size);
}
...
```

The doStep function did not change except for the fact that it was run on a GPU instead of on the CPU. This allowed the code to run each set of doStep instructions at the same time and therefore take roughly 1/512th of the time it would have otherwise taken.

3.2 Scene Completion Integration

Utilizing the object per-pixel labeling system that the UC Berkeley lab put out, I devised a system that would be able to capture a scene from a Kinect camera and output a series of pointclouds for detected objects within a cluttered scene. The overview of the algorithm is as follows:

```
Data: PointCloud
Result: Segmented pointclouds for each detected object
objects = fcnn(PointCloud).getAllObjects();
foreach object in objects do
    completedShape = shapeCompletion(object);
    grasp = planGrasp(completedShape);
    executeGrasp(grasp);
end
```

Algorithm 1: Find all objects in a scene

In order to segment the scene pointcloud a rough per-pixel labeling is obtained from the fcnn. This will be a little noisy and will likely not translate perfectly onto the scene as some examples have shown. Once we have a rough per-pixel labeling we can use supervoxel clustering to find patches of the pointcloud that are likely clustered together and then find the average object id for that given cluster. We can perform this operation for each cluster in the scene and then merge similarly labeled nearby clusters.[1]

[1]http://pointclouds.org/documentation/tutorials/supervoxel_clusteri

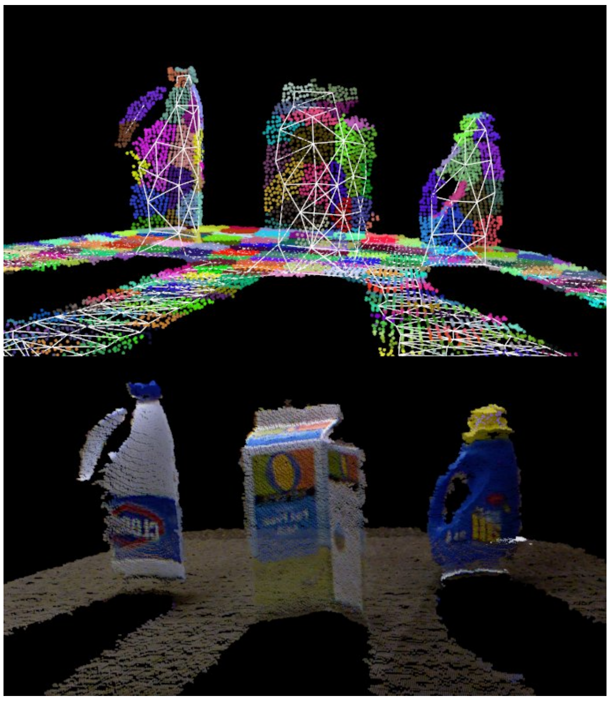

Figure 2: An example of supervoxels and adjacency graph generated for a cloud

Once each of these clouds are properly labeled and merged, we can filter by the pointclouds that only have the object tag. This will give us a much larger point cloud from the entire scene that contains any object point cloud. We then apply the euclidian clustering algorithm to segment the scene into separate pointclouds each representing a singular object. Each of these partial meshes are then passed into the shape completion pipeline and a full object with a pose is generated. This gives the system fine grained control over which object to then choose for the pick and place pipeline.

Note that this code has not yet been written and a lot of the work for this has been purely experimental. One step of the fcnn is to generate an HHA formatted file from the depth image captured from the Kinect. The HHA format for the NYU depth database[1] is manually touched up to be more accurate, however the system we are trying to design is much more dynamic. We decided to test whether the HHA format is still accurate given just the depth image without a touched up representation. In order to do this we took the matlab saveHHA code that was provided by UC Berkeley and modified it so that it took the same raw depth image as both arguments instead of one raw and one touched up. For results comparing the analysis of this section of code see the next section. Once we have this pipeline fully configured, we can then take the fcnn and use it on a raw pointcloud from the Kinect.

There are some artifacts that are held over from using the raw information - but the resultant per-pixel labeling is only slightly worse than that of the touched up version.

3.3 Depth Rendering

The depth rendering pipeline was a system that took a series of object files and rendered the depth image of the object. Given that there are several thousand objects and for each object we want several different views of the same object, the faster the depth renderer goes the better. Currently the system uses Gazebo, a robot simulation platform[2], to generate all the depth images which can take a few seconds per image. Instead I decided to streamline the process by building a standalone, headless depth renderer that uses OpenGL to generate the depth images.

[2]http://gazebosim.org/

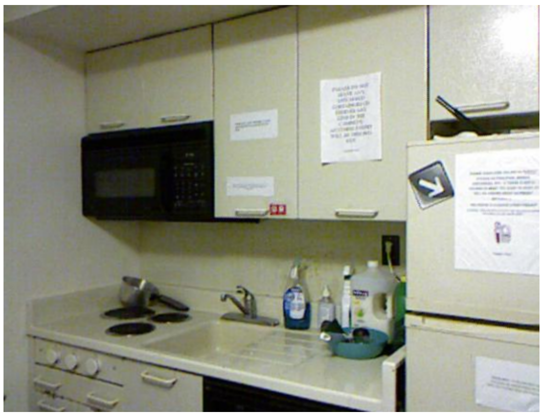

Figure 3: Actual RGB image being processed

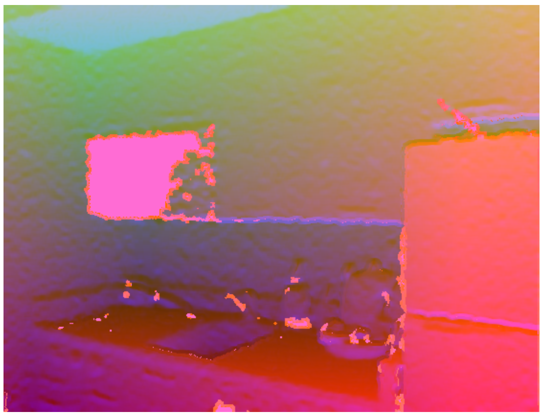

Figure 4: HHA image from the raw depth information. Artifacts from some black colored or dark surfaces where depth was not accurately captured.

In order to build this system I took a simple OpenGL 2D renderer as a base that reads in an obj file and outputs a screen with a view of the object model as it is supposed to be displayed. This view can be tilted or panned to view the object at different angles. I then took the code for manipulating the object in space and randomly assigned the object a rotation. I set this function to be called within a for loop that runs for approximately 500 times and at each interval I took a depth image of the object as it looked in that pose.

For each of these depth images I stored them in a specific directory to keep them for later use. The major advantage to using OpenGL over Gazebo is that OpenGL gives us far more control over the design of the application and will run a lot faster since it no longer has to create a simulation environment. OpenGL also supports a headless rendering system which makes it even faster.

```
...
while (_model->hasMoreSnapshots() &&
↪   !glfwWindowShouldClose(_window))
{
            glClear(GL_COLOR_BUFFER_BIT |
            ↪   GL_DEPTH_BUFFER_BIT);

            _model->update();
            _model->reload();

            _model->screenShot(0);

            glfwSwapBuffers(_window);
            glfwPollEvents();
}
```

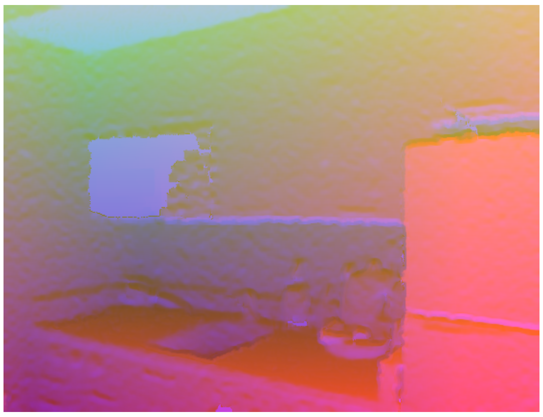

Figure 5: HHA generated from touched up depth information. The artifacts are no longer there.

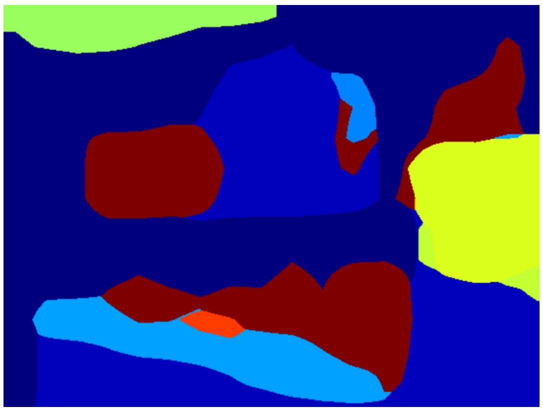

Figure 6: Per-pixel labeling of the image using the touched up version of the hha image. Very clean and the object labeling in red makes sense given the scene.

. . .

Theoretically there are additional performance gains that could be attained through writing a CUDA based depth renderer and removing the need for the entire OpenGL pipeline, but because the OpenGL pipeline is a very well optimized and hardware accelerated system it didn't make sent to do this.

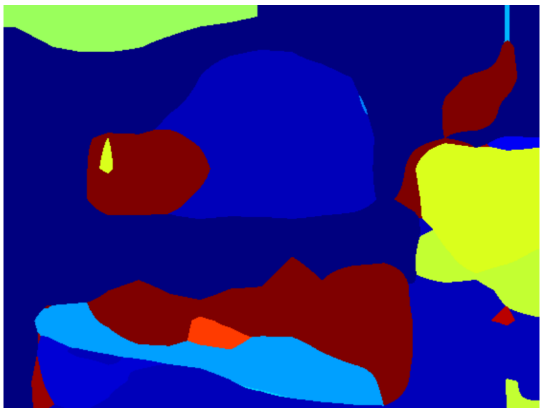

Figure 7: Per-pixel labeling of the image using the raw depth hha file. Less organized but would be cleaned up by using a supervoxel algorithm.

Figure 8: OpenGL rendering a cat object model

4. RESULTS AND ANALYSIS

4.1 Experiments

4.1.1 Shape Completion Testing

In order to affirm I had written the correct code, I took sample meshes from the shape completion pipeline code and compared the CPU results with the GPU results. I found that there was no difference in the results. I also found that the code was able to run an order of magnitude faster. The code that ran on the CPU was taking on average 8.3 seconds to run on a quad core Intel CPU. I was able to get the runtime of the shape completion framework down to an average 1.2 seconds on the same CPU using a GeForce GTX 780 Ti GPU.

4.1.2 HHA Comparison

The raw HHA files differed from their touched up counterparts by roughly 27% on average by comparing the pixel values between the two images. The resulting per-pixel labeling showed a difference of roughly 80 pixels on average between the two, which was a great result considering that there was no human intervention to achieve this result. With additional cleaning this result would likely become more accurate than it is now.

4.1.3 Depth Rendering

By creating a new depth rendering system I was able to reduce the time spent rendering from several days to only a few minutes. The new rendering platform can achieve a framerate of 120fps, which is 240 times faster than the currently existing Gazebo system. However there are some bugs with the implementation so it can only render 2D images at the moment. More testing is necessary to fix the errors with OpenGL and depth rendering.

4.2 Next Steps

The next steps would be to fully implement the scene segmentation pipeline and take the code for the depth renderer and fully flesh it out. A lot of the framework is in place to get these applications implemented but due to time constraints they have not yet been finished.

5. CONCLUSION

In conclusion, these additions to the shape completion pipeline have shown to be promising and with continued effort could be enhanced. The most promising result is the order of magnitude performance increase of the shape completion system. Once the depth renderer is operational the generation of test data will also be much faster.

6. REFERENCES

[1] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In $ECCV$, 2012.

[2] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. $CoRR$, abs/1605.06211, 2016.

[3] Jacob Varley, Chad DeChant, Adam Richardson, Avinash Nair, Joaquín Ruales, and Peter K. Allen. Shape completion enabled robotic grasping. $CoRR$, abs/1609.08546, 2016.