AUTOMATING MODEL CONSTRUCTION FOR SCENE UNDERSTANDING

By

KRISTLE CORY SCHULZ

A Thesis Submitted to The Honors College

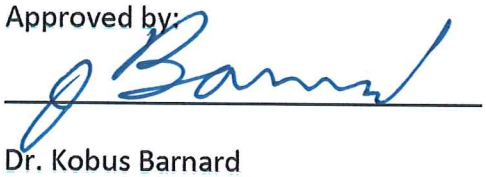
In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

M A Y 2 0 1 4

Approved by:



Dr. Kobus Barnard
School of Information: Science, Technology, and Arts

The University of Arizona Electronic Theses and Dissertations
Reproduction and Distribution Rights Form

The UA Campus Repository supports the dissemination and preservation of scholarship produced by University of Arizona faculty, researchers, and students. The University Library, in collaboration with the Honors College, has established a collection in the UA Campus Repository to share, archive, and preserve undergraduate Honors theses.

Theses that are submitted to the UA Campus Repository are available for public view. Submission of your thesis to the Repository provides an opportunity for you to showcase your work to graduate schools and future employers. It also allows for your work to be accessed by others in your discipline, enabling you to contribute to the knowledge base in your field. Your signature on this consent form will determine whether your thesis is included in the repository.

Name (Last, First, Middle)
SCHULZ, KRISTLE, CORY

Degree title (eg BA, BS, BSE, BSB, BFA):
BS

Honors area (eg Molecular and Cellular Biology, English, Studio Art):
COMPUTER SCIENCE

Date thesis submitted to Honors College:
MAY 7, 2014

Title of Honors thesis:
AUTOMATING MODEL CONSTRUCTION FOR SCENE UNDERSTANDING

The University of Arizona Library Release Agreement

I hereby grant to the University of Arizona Library the nonexclusive worldwide right to reproduce and distribute my dissertation or thesis and abstract (herein, the "licensed materials"), in whole or in part, in any and all media of distribution and in any format in existence now or developed in the future. I represent and warrant to the University of Arizona that the licensed materials are my original work, that I am the sole owner of all rights in and to the licensed materials, and that none of the licensed materials infringe or violate the rights of others. I further represent that I have obtained all necessary rights to permit the University of Arizona Library to reproduce and distribute any nonpublic third party software necessary to access, display, run or print my dissertation or thesis. I acknowledge that University of Arizona Library may elect not to distribute my dissertation or thesis in digital format if, in its reasonable judgment, it believes all such rights have not been secured.

☑ Yes, make my thesis available in the UA Campus Repository!

Student signature: Kristle Schulz Date: May 2, 2014

Thesis advisor signature: J Barnes Date: May 2, 2014

☐ No, do not release my thesis to the UA Campus Repository.

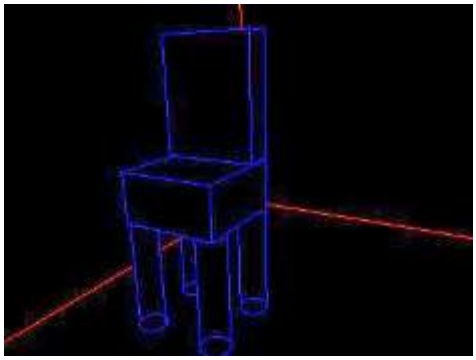
Student signature: _____ Date: _____

Last updated: 04/01/13

Abstract:

The Vision Department at the University of Arizona seeks to teach robots how to distinguish pieces of furniture in a two-dimensional image scene. Currently the robot only identifies a small number of furniture pieces. Before we can teach the robot to identify a new object, we must first build a heuristic with a list of dimensions and part ratios. To do this now, users must have an in-depth understanding of the existing room code as well as the format of the definition file.

I was tasked with a writing a GUI (Graphical User Interface) that will allow users of any technological background to design a new piece of furniture. The GUI then renders an image from the data the users enter. For example, the GUI was able to generate the chair seen in Figure 0. While this chair seems rather ordinary, it is proof of concept that our GUI is compatible with the existing software. Users can then alter the parameters and produce images that are not as "normal". Once the users are satisfied with their object, we will take the new heuristic and add it to the existing software, to see if a machine can recognize the new pieces of furniture.



(Figure 0). A standard chair rendered from the information collected in our GUI.

Introduction:

The field of computer vision centers on giving machines the ability to interact with the world in a human-like manner. To do this, robots need to be able to differentiate the objects in an image, identify these objects so the robots can gain a sense of their purpose, and be able to interact with the objects appropriately. Essentially, robots need to understand a three-dimensional scene from a two-dimensional image, in terms of the properties of the present structures (What is Computer Vision?).

The applications for a robot that can interact correctly with the world are endless. Just some of the fields that would benefit from this autonomy are manufacturing, medicine, transportation and security. If robots could understand when something is going wrong and have the ability to discern the correct procedure for fixing it, we could teach them to perform menial jobs such as assembly or product-verification. We could train robots in obstacle avoidance so they can better assist and serve humans. We might even be able to go as far as making the NASA rover self-driving, since it would have the ability to identify boulders and crevices and know to avoid them, thus prolonging its mission (What is Computer Vision?). A completely autonomous being is far in the future, but current researchers have made great strides in teaching machines to recognize and identify different objects.

Background and Current Work:

Universities all over the world are approaching the subject of computer vision in very different manners. Researchers at Stanford University have taken their work outdoors in an attempt to teach a robot how to identify sky, roads, buildings, mountains and more (Gould, Fulton and Kolle). Researchers at Carnegie Mellon are focusing on teaching a robot how to interact with previously identified objects in a room, in the sense that, once a robot can identify an object, it should be able to know how humans would interact with it (Gupta, Satkin and Alexei). Faculty at Cornell are attempting to have robots identify human body parts, such as arms and legs, in a variety of different positions, including standing, sitting, and slouching (Huttenlocher). Each of these research projects is discovering their own set of inconsistencies, such as long, windy roads being interpreted as rivers, which leads to cars in the same image being interpreted as boats; or background lines running behind a human being mistaken for arms. No approach has been perfected but each is making amazing bounds in expanding the limits of what a machine can learn.

Here at the University of Arizona, Dr. Kobus Barnard and his team are teaching a machine to recognize different pieces of furniture in a room scene. This process currently involves a robot viewing a scene with a camera and outlining the objects it is able to recognize (Figure 1).

Before we get to the recognition step, however, a machine must first be able to recognize the borders of the room, or the "room box", which is defined by the corners, the juncture where the ceiling meets the walls, and the juncture where the walls meet the floor. This allows the machine to hone in on identifications since pieces of furniture in the same category are typically found in particular locations in a room. For example, couches are low to the ground and do not take up more than half of the wall, while pictures are typically found at a midpoint on a wall and are not extremely large. Once a room box has been established, the machine can work on separating the different objects in the room. After it has done that, the robot takes into consideration the location of the object in relation to the room box, as well as its relative scale to the room itself. It can then compare this information to its heuristics and see if any matched (Pero).

(Figure 1). The robot will "see" the room through its pinhole camera then attempt to outline all the pieces of furniture it can recognize.

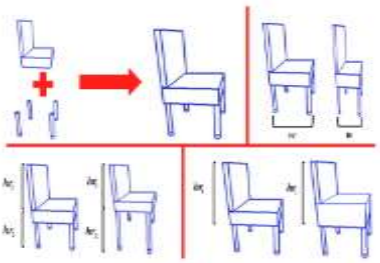
Right now, the heuristics are relatively simple in that they don't take into consideration modern or unusual forms of a piece of furniture. For example, chairs must have four legs and no armrests, while tables are expected to have four legs and a square top. Once the machine can successfully identify these simplistic forms of furniture, we need to be able to expand the current heuristics so that the robots can recognize such pieces as three-legged chairs, rolling chairs, or circular tables. This will involve adjusting priors on size and typical location as well as creating a series of ground-truth images for these new pieces of furniture. A ground-truth image is one that has been manually outlined by a human and is considered to be completely accurate.

Currently, the researchers pass the existing software a two-dimensional room scene, which is what the camera of the robot would see. The software returns an image in which the objects it has been able to identify are outlined in different colors. In this image, each color is representative of a different category of furniture. As we can see in Figure 2, beds are outlined in blue while pictures are outlined in green. Now, the newly marked image is compared to the ground-truth image. Then, analyses are run on the differences between the two images to give the machine a score, which represents how successful it was in identifying the objects in the room (Pero).

(Figure 2). A sample image returned by the modeling program with notation that the picture frame and bed were recognized (Pero).

In the current heuristics, a piece of furniture is broken into its distinct components. For example, a chair is built from a set of legs, plus a seat and a back (Figure 3). Each one of these

components has a series of dimensions, which are either absolute (fixed) or can fall within a valid range (not fixed). Currently, a researcher who wants to provide a heuristic must manually enter these values into a text file, in a format that can be correctly parsed by the software.



(Figure 3). A chair being broken down into its two main parts: an L-shaped back and seat, and its four legs.

The goal of this honors project was to abstract out the technical details and create a user-friendly GUI (Graphical User Interface) which will allow the users to determine whether or not the values they have entered will build the shape they desired. The GUI does this by using the data the users enter to render a three-dimensional image.

User's Manual for the GUI:

Our GUI reads in a text file that enumerates the possible parts and the number of parameters or dimensions for each part. For example, we would want three parameters for a chair back—length, width and height—while for a chair leg, we might only need two—diameter and height. This file will only need to be updated if a new type of part is added (ie: a part that has three legs instead of four).

The GUI is run in a terminal and it takes at most two command line arguments: `./object_builder [infile] outfile`. The infile argument is optional and allows users to import configurations they were previously working on. If an infile is specified, our GUI will parse it and build instances of the parts, based on the data from the infile. If an infile is not specified, we will start a new configuration from scratch.

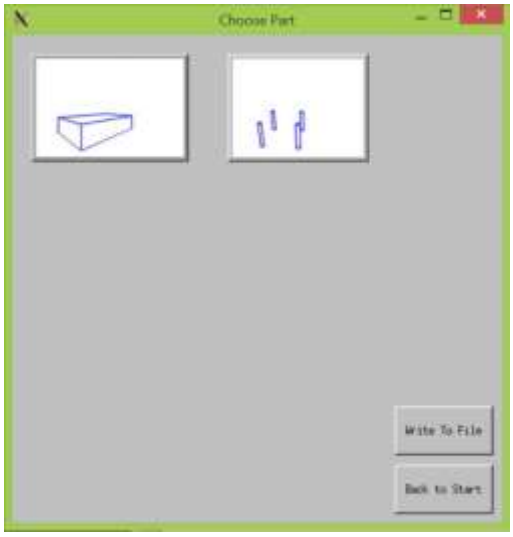
Regardless, the first screen the user will see is the start menu (Figure 4). If the user specified an infile, he or she will also see a dropdown menu, from which the user can select one of their existing parts to either edit or delete entirely (Figure 5).

(Figure 4 left). This is the first screen a user will see when opening the GUI for the first time.

(Figure 5 right). If the user has already entered data for a part, they have the ability to edit or delete that part through the dropdown menu.

When the user clicks “Add Part”, they are directed to the Part Menu, which consists of a series of buttons with image icons (Figure 6). Each button represents a part description in the initial file our GUI read in. There are six main parts that we can use to build our objects: a single block, a grouping of four legs, three L-shaped blocks that differ in the height of the back and the attachment point, and a single leg (Figure 12).

If a user clicks on one of these buttons, they are directed to another window containing an image of the part for which the user is now entering a data, a text box for each parameter (this number was specified in the original description file) and a dropdown menu symbolizing whether this number is fixed or it has a range of acceptable values (Figure 7). If a parameter can have a range, two more text boxes appear, symbolizing the minimum and maximum values that this parameter could take (Figure 8). The user will fill out all the necessary parts then hit “save”. At this point, the GUI will perform minor error checking to ensure the parameter sizes are non-negative and, if range values are listed, that the parameter listed falls within the given range.



(Figure 6 left). Our Start Menu, or Parts Palette. At the beginning of our project, these two were the only parts a user could choose from. (Figure 7 above). Once the user clicks on a part, he or she must provide data for the number of parameters, in this case, two values.

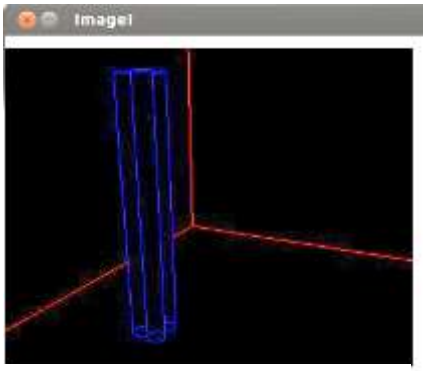
(Figure 8 below). If the dimension can have an acceptable range of values, the user will provide the minimum and maximum values. (Figure 9 right). If more than one part is created, we will ask the user for the height of the new part and use the sum of the user entered values to dynamically adjust the height of the first part.

The GUI must also record the height ratio of each individual part to the entire object so the final image can be rendered properly. We will ask the user for the decimal representation of this ratio. If the user only creates one part, the height ratio is automatically set to 1.0. If the user chooses to add more parts, however, the previous heights will be adjusted according to the new height a user enters (Figure 9). Error-checking will be done to ensure the entered height is less than 1.0.

(Figure 10). A sample text file produced by saving our configuration in the GUI.

Now, the GUI writes the information the user just entered to a text display that shows the user what parameters they listed, how many parts they have added and the height ratio of each (Figure 10). This information is also written to a text file, the name of which is specified by the user when running the object_builder.

Then, that text file is passed as input to the existing rendering software and a three-dimensional representation of the data the user entered is produced. This image is displayed in a separate window and is updated every time the user hits save (Figure 11).



(Figure 11). A sample rendering of four legs produced by the data from the GUI.

At the point, we return to the Start Menu, where the part the user just entered has been used to populate a dropdown menu of existing parts (Figure 5). From here, the user can continue to build new parts but can also edit parameters of existing parts or delete parts entirely. When they are finished, the text buffer is automatically updated to reflect these changes. Again, in future iterations, we plan on removing the text buffer and just producing an image, so the user can see what their design truly looks like and how its different parts are proportioned.

Technical Details:

Our GUI is written using FLTK (Fast Light ToolKit) which is a cross-platform C++ GUI toolkit. It is similar to Java's SWING in that it has input boxes (FL_Input), placeholder boxes which can be filled with anything (FL_Boxes), buttons (FL_Button), text displays (FL_Text_Displays), text fields (FL_Text_Buffers, which will be put inside our FL_Text_Display), images (FL_JPEG_Images), and a dropdown menu (FL_Choice).

One of the main differences between FLTK and SWING is that callbacks (the equivalent of Java's ActionListeners) are silently passed two parameters: the first is an Fl_Widget, and the second is a void *. Here, the FL_Widget is a reference to the item that was clicked on. We attach callbacks like this

```
aButton->callback(writeToFile);
```

What this means is that the writeToFile callback method is run if the button, aButton, is clicked on. Inside that specific method, the FL_Widget parameter can be cast to a button, if we needed to know what object invoked the callback.

If we need the callback to have an additional piece of information, we can say

```
range1->callback(rangeValuesCB,&array[inputCounter*2]);
```

where rangeValuesCB is invoked when the text field, range1, is used. We are also passing the method an element in an array of numbers, which we can access by casting the second parameter, a void *, to an int *.

What was initially difficult about these callbacks was that you cannot invoke them without clicking on an Fl_Widget. So, if I had error-checking code in a callback that I wanted to use elsewhere, I cannot invoke the callback from my program because I do not have the ability to silently pass the requisite parameters. My file therefore became more cluttered as I was forced to write a number of helper methods that could be called both in the callbacks and the rest of my code.

1. Data Types

We maintain three vectors in the GUI: one is a vector of Parts, which is built from the initial text file which specifies the possible parts and the number of dimensions for each; the second is a vector of Instances, which are the parts the user has enumerated; and the third is a vector of Heights where each height is a ratio of a part's height to the height of the entire object. The Height and Instance variables coincide in the sense that the first element in the Height vector is the height of the first instance in the Instance vector.

A Part consists of three pieces of information: a filename, which is a String that represents where the image of the part is located; the number of parameters, which is how many values the user will be specifying; and the part number, which will range from 0 to n so we can associate instances with their respective part.

An Instance consists of four pieces of information: a part number, so we can associate this instance with a part; the parameters, which will be a vector of floats, representing the

values the users enter; a vector of Booleans that corresponds to the parameters vector and says whether that particular value is fixed or not; and finally, if the parameter value is not fixed, the range of values that are acceptable. The ranges are stored as a pair and default to 0 is the parameter is fixed.

2. Writing To A File

When the user hits save, we will iterate over the Instances vector and produce a text file. The format looks like this:

num_parts: 2
part_indexes: 0 1
num_parameters: 3
parameter_means: 9 5 1
parameter_ranges: 0 0 0 0 0 0
fixed_parameters: 0 0 0
num_parameters: 2
parameter_means: 9 7
parameter_ranges: 0 0 0 0
fixed_parameters: 0 0
part_heights: 0.7 0.3
part_heights_ranges: 0 0 0 0
part_heights_fixed: 1 1

In this file, we're building an object made of two parts where the first part has three fixed parameters (values 9, 5 and 1) and will take up 70% of the total object height. The second part has two fixed parameters (values 9 and 7) and will take up 30% of the total object height. If we assume that this object was produced from the configuration in Figure 6, the first part would be a rectangular block and the second part would be a set of four legs.

In general,

- "num_parts" are the total number of parts that compose the object the user built
- "part_indexes" are the indices of the parts chosen. Each part is represented by four pieces of data: "num_parameters", "parameter_means", "parameter_ranges" and "fixed_parameters". Therefore, the first block of "num_parameters" through "parameter_ranges" corresponds to the part at index 0, while the second block corresponds to the part at index 1.
- "num_parameters" is how many dimensions the part has
- "parameter_means" are the dimensions entered by the user for each instance

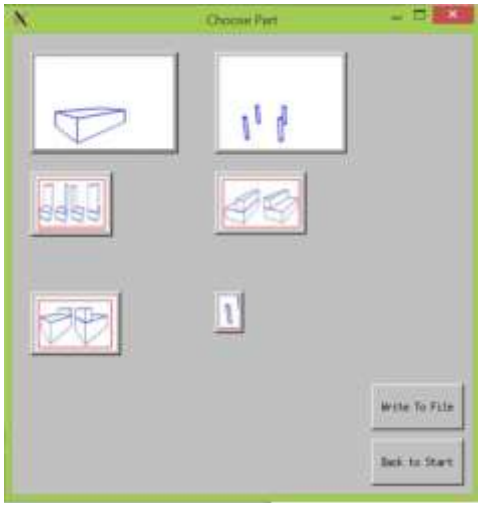
- "parameter_ranges" detail the minimum and maximum value for a parameter, if it is determined that it is acceptable for this parameter to have a range.
- "fixed_parameters" are whether or not the corresponding parameter for the instance is fixed (1 is it is, 0 if it is not)
- "part_heights" correspond to how tall the part is, where 1.0 is the height of the entire object.
- "part_heights_ranges" details whether or not a part is allowed to be taller or shorter than the mean value the user just provides
- "part_heights_fixed" specifies whether or not a part must always appear at the same height, or whether it has a range of acceptable values.

The ability to write to a file serves a two-fold purpose. One, it serves as input to the existing modeling program which allows us to display a rendered image based on the data the user provides, so the user can see whether they entered their dimensions correctly. Two, the file also gives the user the ability to save their work and come back to it later since the GUI has the ability to read from an infile and restore the existing configuration.

Opening an existing configuration is simple in concept since we already have methods that traverse our Parts vector. Therefore, it is just a matter of populating our Instances vector from the existing file configuration, then calling the relevant methods to adjust the welcome window for the existing parts. Though this information will be valid, in the sense that a parameter must fall within the given ranges, it may not be complete (ie: not all of the fields may have been filled out).

3. Calling the Modeling Program

In order to call the existing modeling program, we must first make sure the parts in our original "file.txt" (the file we read from to determine the viable parts and the number of parameters for each) are in sync with the modeling program. Therefore, we must adjust "file.txt" so includes our six viable parts (a block, a set of four legs, three different variations of a back connected to a seat, and a single leg) and correctly enumerates the required number of parameters for each.



(Figure 12). Our Parts Palette adjusted for the 6 parts the existing software knows about. The parts (L-R) are a block, a set of four legs, three different variations of a back connected to a seat, and a single leg.

Once we've done that, our parts palette will display all six parts in the correct order (Figure 12). Our next step is creating a furniture model that the modeling program can read in. To do this, we must first build

Room_object_part_options for each part instance. A Room_object_part_option takes three parameters: a kjb::Vector representing the mean value for each parameter, a std::vector of kjb::Vectors representing the minimum and maximum values for each parameter, and a std::vector of Booleans representing whether or not a parameter is fixed. A kjb::Vector is an existing user-defined class that is compatible with the kjb::workspace, or the workspace that is used by the room software. We declare a kjb::Vector of size 3 by saying

```
kjb::Vector one(3);
```

Then we can fill it with elements by saying

```
one(0) = 1.5;
```

This means fill the first element of our kbj::Vector, named "one", with the value 1.5.

As seen in the "Data Types" section, all of our vectors are standard vectors (std::vectors), meaning we must iterate over all our Instance vectors and convert the individual parameter vectors into kjb::Vectors. All this entails is creating a kjb::Vector of the length of our parameter std::vector, then copying over the values from the std::vector. The range is a little different since the room_object_part_option requires a std::vector of kjb::Vectors. This means we will make a kjb::Vector of size two and fill the first element with the minimum parameter value, and the second with the maximum. Then, we will add this kjb::Vector to the std::vector.

Once we have our Room_object_part_options, we can then build a Room_composite_object_options, which takes in six parameters: the number of parts in the object we want to model, a std::vector of integers representing the indices of each part, a std::vector of Room_object_part_options (the objects we were building in the previous paragraph), a kjb::Vector representing the heights of the parts, a std::vector of kjb::Vectors representing the minimum and maximum values for the heights of each parameter, and a std::vector of Booleans representing whether or not the heights are fixed.

Once we have our room_composite_object_options, we can call the rendering program.

4. Displaying the KJB Image

There is a small incompatibility issue with the KJB Buffered Image that the modeling program produces. FLTK is not designed to be able to read that image extension, so we must convert it to an Fl_RGB_Image in order to display it in our GUI. A Fl_RGB_Image takes a pointer to an array of characters (a datatype which has an integer backing) as its parameter, as well as the width and height of the image. We can get the width and height easily, as a KJB image has get_num_rows() and get_num_cols() methods. So all we need now is to convert the KJB image into an array of chars.

We have the ability to break down a KJB Image into its individual Pixels. A Pixel is an existing user-defined structure made up of three floats, which represent its red value, its green value and its blue value. Therefore, we can get a 2D array of Pixels from the KJB Image the modeling program produces, and iterate over it, adding first its red, then its green, then its blue value to the array of characters.

Let's take an example. We have an array of

AB
CD

where A, B, C, and D are Pixels. If we denote AR to be the Red value of A, AG to be A's green value, and AB to be A's blue value, our 1D array of chars would look like:

AR AG AB BR BG BB CB CG CB DR DG DB.

Now we can pass this array to our Fl_RGB_Image constructor and then display the result in our GUI. From here, the user can see what configuration their input produces and figure out whether or not it looks as intended.

Results:

It turns out that the act of calling the rendering software from the GUI was more difficult than we originally thought. Therefore, the project had to be adjusted so it could be completed by the end of this semester.

In the most recent iteration, the GUI calls a derivative of the original rendering software called the room_drawer, which will not take into consideration the possible ranges of a part's dimension. We will call the room_drawer from our GUI by invoking a system call, which allows us to run another C++ program, and passing it the requisite four arguments: the text file our program outputs; the specifications for the camera angle (remember that the robot is viewing these scenes through a camera); a solid black image which is the background over which our object will be rendered; and a file to which our new image will be written.

The room_drawer takes this information and renders an image which it writes to `temp.jpg`, the filename we will specify in the system call. Our GUI will then make an FL_JPEG_Image out of `temp.jpg` and display it in a separate window. We will call the room_drawer every time the user clicks "save" so he or she can see how their object is progressing (Figure 11).

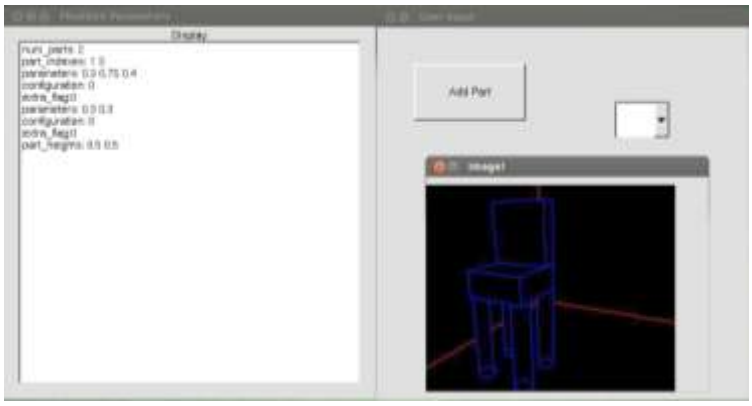
In order to call the room_drawer, we had to make a few changes to the text file that was produced. So our output file now looks something like this:

Type: N3kjb20Parametric_parapipedE
width: 13.6975
height: 4.92147
length: 13.8558
pitch: 0
yaw: 1.04275
roll: 0
centre: 2.57654 -0.0899965 1.97199 1
num_objects: 1
object_type: 1
center: 3.41009 -1.54527 -5.45818 1
bounding_size: 0.985415 2.85092 0.961112
surface_attachment: 4
num_parts: 2
part_indexes: 1 2
parameters: 0.3 0.75 0.4
configuration: 0
extra_flag:0
parameters: 0.3 0.3
configuration: 0
extra_flag:0
part_heights: 0.5 0.5

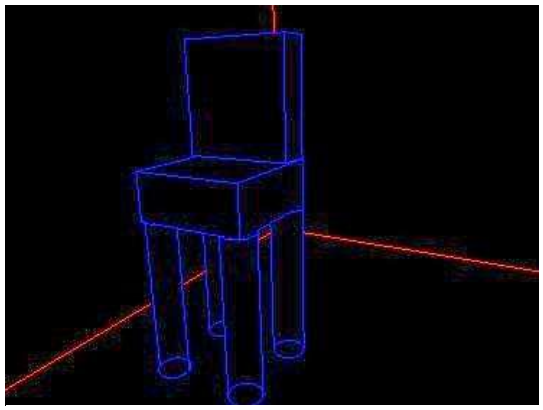
The first thirteen lines are constants for the image rendering software and are not changed from file to file. Everything from “num_parts” onward is still determined by our GUI, with the exception of the “configuration: 0” and “extra_flag: 0” fields, which are another set of constants. As you can see, we only kept the parameter_means field and ignored whether a part was allowed to have a range of values.

We are still writing the text in the format specified in Figure 10 to the file the user specifies when running the GUI. Now, however, our GUI writes the adjusted format to a second file, testroom.txt Therefore, we can run the room_drawer by passing the following String as an argument to our system call:

```
/room_drawer testroom.txt testcamera.txt blackimage.jpg
./temp.jpg
```

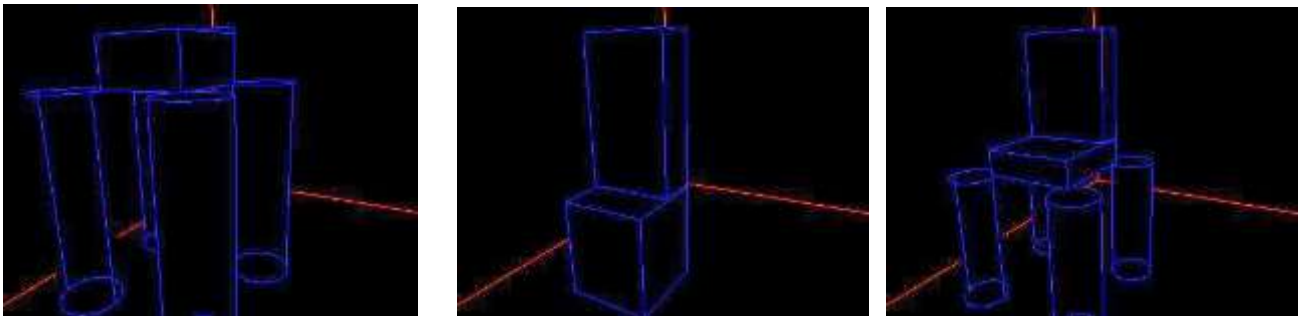


(Figure 13). A screenshot of the GUI producing a chair from the data the user entered (also visible in the text field) and displaying it in a separate window.



(Figure 14). A close-up of the image rendered from the information in Figure 13.

After entering the data from file included on page 13, we were successfully able to produce a chair (Figures 13 and 14). However, not all of our renders were as "normal". Depending on the parameters entered, we could generate some rather unusual shapes (Figure 15).



(Figure 15). Other pieces of furniture we were able to build with the GUI.

Future work:

While we able to achieve our goal of rendering an image from user-defined data by calling the room_drawer, our GUI would be more complete if it rendered the data independently, using the Room_composite_object_options and the KJG to Fl_RBG_Image converter we built. Besides avoiding a frowned-upon system call, it would allow our GUI to be an independent module.

The room_drawer also ignores the ranges we gathered if the parameter is not fixed. It would be useful to produce two separate images: one rendered with the minimum values the user provided and one rendered with the maximum values. This way, we can ensure the user deems both the smallest and largest values for any instance acceptable. It might also be

interesting to ask for a standard deviation instead of a minimum or maximum value, to see if those two sets of data would produce different images.

Also, for those who aren't familiar with the specifics of each dimension (ie: how large a particular dimension actually is), our GUI could be slightly clunky or hard to use correctly. Therefore, we could abstract out even more of the technical details and provide for them a way to graphically size their object. We could do this by building an OpenGL program that will allow the user to dynamically adjust the size of the parts until the object looks visually correct, without needing text fields. In this though, we must remember that object only interact at certain points, for example, legs only connect on the corners of a typical table, so altering the table top would also involve adjusting the relative size of the legs. This way, the user doesn't have to "guess" while entering values and hope that their finished image will turn out like they had intended. We hope that this new code will be more user-friendly and allow the user to build an object of the correct size for the room project to analyze.

Finally, we want to see if our new object can be identified by the existing recognition software. We would want to add our new heuristic to the database then stage some scenes with the new object in them. We would pass these images to the room project along with some ground-truth images, to see if the project software was able to recognize any of our new objects. If it was able to recognize only some of our new objects, we would need to redefine our heuristic to adjust for the pieces that were not detected. If the software was not able to recognize any of our objects, then this example might serve as a future work for those working on the project software, as long as the object that was built is one that realistically might be seen in the real world.

References:

Gould, Stephen, Richard Fulton and Daphne Kolle. "Decomposing a Scene into Geometric and Semantically Consistent Regions." *ICCV* 2009: 8. electronic.

Gupta, Abhinav, et al. "From 3D Scene Geometry to Human Workspace." *Computer Vision and Pattern Recognition* (2011): 10. electronic.

Huttenlocher, Dan. *Computer Vision for Recovering Information About Scene Geometry*. Cornell University, 2012. electronic.

Pero, Luca Del. "Understanding Bayesian rooms using composite 3D object models." *Computer Vision and Pattern Recognition* (2013): 8. electronic.

"What is Computer Vision?" 23 January 2014. *Rensselaer.* electronic. 20 April 2014.