RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

SENIOR CAPSTONE DESIGN, SPRING '13

Computer Vision-Based 3-D Reconstruction for Object Replication

Authors:
Ryan CULLINANE
Cady MOTYKA
Elie ROSEN

Advisor:
Professor Kristin DANA

May 1, 2013

Abstract

The Microsoft Kinect for Windows has proven to be a valuable tool in the field of computer vision. The Kinect is comprised of an infrared laser projector and depth sensor. The depth data of a scene is run through a bilateral filter and vector mathematics is used to define the coordinates, connecting lines, the vertices, and edges to form a three-dimensional mesh. The software displays the raw depth data and infrared camera image, this allows the user to filter out objects closer or further than a specified depth, and exports the reconstructed three-dimensional mesh. That mesh is then sliced into horizontal layers and converted into G-code, a machine language that maneuvers the 3-D printer where to extrude the ABS plastic to create a physical replica of the reconstructed object.

Contents

1 Introduction

The Computer Vision–Based 3-D Reconstruction for Object Replication is accomplished by using a Kinect for Xbox. Originally, the Kinect was created for entertainment, but in recent time, it has been introduced to the field of robotics and computer vision. The Kinect is a quick, reliable, and affordable tool that uses a near-infrared laser pattern projector and an infrared camera, as well as a depth sensor that relays three-dimensional data through the Kinect Software Development Kit (SDK).

The robotics field is beginning to find an increasing number of uses for the 3-D printer. The most innovative aspect of the 3-D printer is the ability to print an object, regardless of interconnecting internal components, and to have it function as intended. For example, any connecting gears that are printed with the 3-D printer will in fact turn as they are supposed to.

2 Methods

2.1 Calibration

The Kinect can be calibrated in a way similar to other cameras for computer vision, the only difference is that changes in the depth have to be present with the pattern in order to calibrate the depth camera. The Kinect can be calibrated with an image of a checkerboard pattern.

2.2 Stereo Reconstruction

Once the Kinect has been calibrated, all that is needed for the stereo reconstruction is a triangulation of viewing rays.

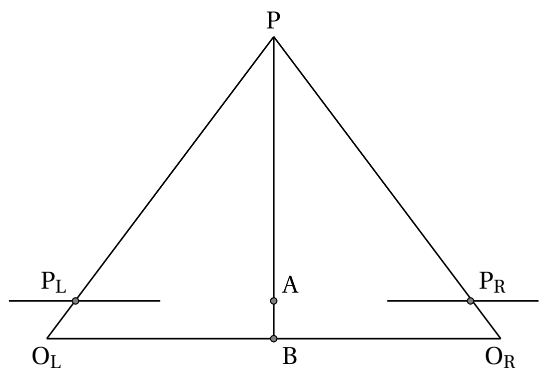


Figure 1: Visual Representation of Depth Disparity.

P is the location of the object in the world, O_L and O_R are the left and right camera centers, P_R and P_L are the appearance of the point P in the two image planes where,

$$P_L = \begin{bmatrix} x_L \\ y_L \end{bmatrix}.$$

The distance between O_L and O_R is T, the distance between the left and right camera. The distance between A and B is the focal length of each of the cameras. If the distance between P and B is defined as distance Z, the following equation can be used to represent the ratio between T and Z, using the theorem of like triangles,

$$\frac{T}{Z} = \frac{T + x_L - x_R}{Z - f} \, or \, \frac{T - x_R - x_L}{Z - f}.$$

Cross multiplying these equations results in,

$$\frac{Z(T - x_R - x_L)}{Z - f} = \frac{T(Z - f)}{Z}.$$

These calculations show that depth, Z, is inversely proportional to the disparity. Therefore,

$$P_L = \frac{f^L P}{Z_L} \text{ and } P_R = \frac{f^R P}{Z_R}.$$

Once there is a corresponding point pair for P from the two images, an algorithm would undo the scale and shift of the pixel points in order to obtain two-dimensional camera coordinates. The midpoint algorithm is then used to find the actual three-dimensional world coordinate that corresponds to that point pair.

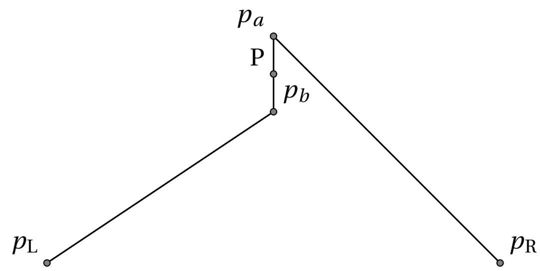


Figure 2: Visual Representation of the Midpoint Algorithm.

Above are the rays $\overrightarrow{\mathrm{O_R} p_\mathrm{R}}$ and $\overrightarrow{\mathrm{O_L} p_\mathrm{L}}$. The line connecting the two vectors, perpendicular to both, is obtained by taking the cross product of these two vectors. The vector,

$$\overrightarrow{p_\mathrm{L} p_b} = a\vec{p}_\mathrm{L},$$

and since point p_R is distance T away from p_L,

$$\overrightarrow{p_\mathrm{R} p_a} = b^\mathrm{L}\mathrm{R_R}\vec{p}_\mathrm{R} + \mathrm{T}.$$

The segment connecting these two vectors can be represented as,

$$\overrightarrow{p_a p_b} = c\vec{p}_\mathrm{L} x^\mathrm{L}\mathrm{R_R}\vec{p}_\mathrm{R},$$

where a, b, and c are unknown constants that can be solved using the three aforementioned equations. The point P lies on the center of this line and is found where,

$$^\mathrm{L}\mathrm{P} = a\vec{p}_\mathrm{L} + \tfrac{c}{2}\vec{p}_\mathrm{L} x^\mathrm{L}\mathrm{R_R}\vec{p}_\mathrm{R}.$$

In order to get the world point M, $^\mathrm{L}$P would is divided by the intrinsic and extrinsic matrices of the Kinect.

Figure 3: The Kinect for Xbox. The hole on the left is an infrared light source, the center is a RGB camera, and the three-dimensional depth sensor is on the right. In addition to these cameras, the base has a motorized tilt and a multi-array microphone that goes along the bottom of the device.

The Kinect accomplishes triangulation by using the known information about the sensor, the data obtained from the infrared projection, and the image received from the camera. The sensor projects infrared light onto an object, the light then bounces back, and the infrared sensor reads in the data from the reflected light. These clusters of captured light can be matched to the hard-coded images the Kinect has of the normal projected pattern, and allows for a search for correlations, or the matching points. While looking through the camera's focal point, the point of interest will fall on a specific pixel, depending on how close or far away it is, providing the end point for the trajectories coming from the camera and projector. These relative lines of trajectories, along with the known information about the distance between the cameras on the Kinect sensor, are used in the triangulation process to find the three-dimensional coordinates of the point. Figure 3 the sensor arrangement of the Kinect for Xbox.

2.3 Bilateral Filter

In order to make the depth data cleaner and less noisy, a bilateral filter is used to remove the erroneous measurements. The bilateral filter takes the depth at every point, and recalculate the depth value at that point based on a waited average of the surrounding pixels in a specified neighborhood. The process takes away some of the sharpness of the depth map, but it removes the noise that skews the results of the three-dimensional reconstruction. The filter follows a Gaussian approach which calculate a new depth at every depth pixel in the image and replaces it with,

$$\mathrm{BF}[\mathrm{I}]_{\mathrm{P}} = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix},$$

multiplied by the neighborhood of the three by three square of pixels around the pixel that is being changed. The resulting depth value represents the average of the nine pixels, where the closer pixels weights in the average are heavier than the further pixel values' weights. [1]

2.4 Mesh Construction

Once the depth data has been filtered, it can be used to create a three-dimensional mesh of the object. At each pixel location two vectors are made, each connecting that three-dimensional point to both the neighboring point to the right and below. Figure 4 shows the square that is is looked at for each x and y coordinate. Cross multiplying the two vectors of adjacent sides results in the orientation vector for the point or its normal vector. As the loop goes through each point, it creates a triangle in three-dimensional space out of the existing points and calculated orientations, which are recorded in three-dimensional point and vector collections. Figure 5 shows the recorded three-dimensional point, normal vectors and orientation of that triangle in the mesh. Each of these points must be added in the correct order, keeping with the right hand rule, so that each reconstructed triangle is oriented in the correct direction. While these three points and vectors are added, separate collections of the indicies and texture information are recorded.

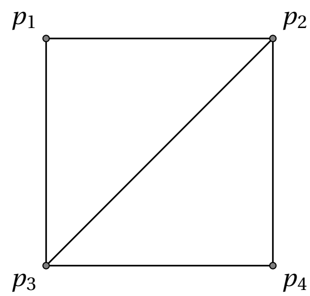


Figure 4: The square used to build the three-dimensional mesh

While reconstructing the front face of the mesh, our algorithm goes through every x and y pixel coordinate starting at (0,0) and ending with (320,240), incrementing by a scale factor number that can be adjusted for less demand on the computers processor. Down sampling for testing was accomplished by setting the number to two so that only every other point was processed. In the software the user can also crop out the left, right, top or bottom of the image. Changing these sliders indicates where on the image the reconstruction of the mesh is going to begin and end. The user can also specify what they would like the minimum and maximum depth to be.

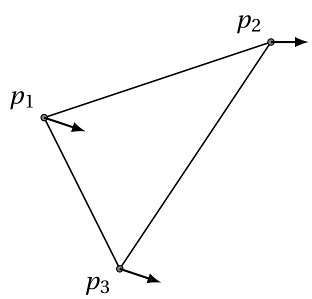


Figure 5: The correctly oriented triangle of the three-dimensional mesh

In order to filter out depth that is further than the intended value, the software looks at the square where the point of interest is the upper left corner. If all four points are outside of the depth range, that point is skipped. If any of the four points on that square are within the depth range, the square is constructed. For the square, any point that is not within the depth range is given the depth of the back wall, the value specified by the user. Lastly, the back wall of the three-dimensional reconstruction is built. A loop goes through every point that is already in the three-dimensional point collection and each point is copied to the end of the collection with the depth changed to the depth of the back wall. The same number of orientation vectors are added to the collection of three-dimensional vectors, each one equaling [0,0,-1]. [2]

2.5 G-code Conversion

The RepRap 3-D printer firmware uses G-code to communicate to the 3-D printer, specifically to define the print head movements. G-code has commands that tell the print head to move to a certain point with rapid or controlled movement, turn on a cooling fan, or select a different extruding tool. Since the RepRap 3-D printer does not have as many features, the G-code generator does not have to add much extra complicated code, but rather only instructions to the printer head and to turn on the heating elements. Since the printer continuously dispenses plastic, it is necessary to find a path for it to take that will build up the reconstructed object layer by layer without placing too much plastic in any specific area. The conversion requires cutting up the reconstructed object into layers and then finding the best path to traverse that layer without overlapping any part of that path. The G-code converter takes in a STL file, cuts it up into horizontal layers, and then calculates the amount of material that is needed to fill each slice. All this is taken care of by the Slic3r project created by Alessandro Ranellucci. [4] An attempt was made to create the converter within the scope of the semester project, however, it was determined that this alone might have taken the whole semester.

2.6 Building a 3-D Printer

The RepRap Prusa Mendel Iteration 2 is nothing more than "an inventory of parts" which lists all of the necessary components and design files for building the printer from scratch. A non-exhaustive list of components include; threaded metal rods, linear bearings, nuts, screws, washers, and electrical heating elements. [6] These parts were purchased from A2APrinter located in Toronto, Canada and arrived by early February. Once the printer arrived, work began to meticulously build the kit following the poorly written specified instructions. In all, there are over 100 pieces to make the printer and each fit in custom 3-D printed parts that were extruded using a secondary MakerBot Replicator 2 3-D printer owned by the Institute of Electrical and Electronics Engineers (IEEE) Princeton Central Jersey Section. The final printer includes a heated extruder nozzle that heats up to 400 degrees Fahrenheit to melt the ABS plastic and a heating bed which heats up to about 200 degrees Fahrenheit to keep the printing object soft and malleable as to allow the completed object to cool evenly since the plastic shrinks minimally when cooling and would cause defiguration in the final printed object.

3 Experimental Results

To obtain an initial grasp of how the Kinect captures data from a three-dimensional scene precompiled demo programs included in the Helix 3D Toolkit were examined. [3] Figure 6 shows the initial tests done using the Helix Kinect Demo which allows the user to control the angle of the Kinect along with other built in settings. This program records the depth data from the infrared depth sensor and maps the color data from the RGB camera to the corresponding points on the depth image to recreate a three-dimensional scene. As can be seen in the figure, the data is very rough and contains a significant amount of missing data causing empty space when viewing the recreation from certain angles. This is due to the fact that only one image of the scene is used and has no methods to obtain information for objects hidden behind closer objects in the scene.

Figure 6: Kinect raw depth data with RGB image mapped to it. It is also important to notice the lack of filtering to smooth the data.

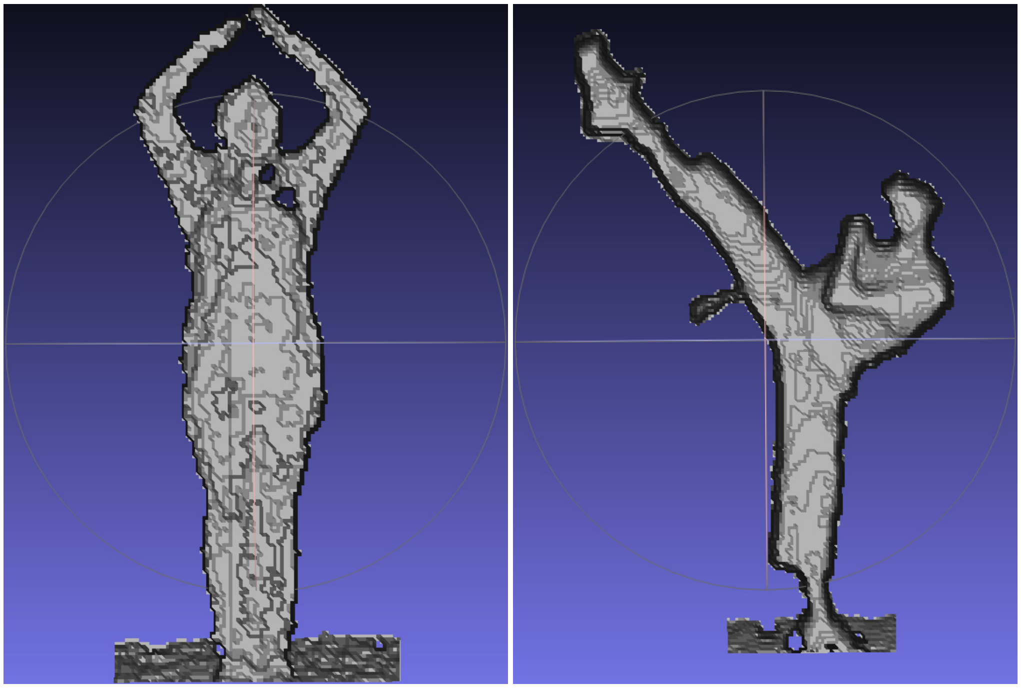
The next step moving forward was to transform this data into a three-dimensional mesh. The basic primitive of a three-dimensional mesh is a polygon, in this case a triangle, which when linked with other polygons creates a contoured surface to represent the object. This representation was initially achieved by creating single triangles at each depth data, which scaled in size depending on the point's distance relative to the Kinect camera as seen in Figure 7. Using this representation as a starting platform, it was quickly determined that much of the data would not contribute to the construction of the model and could therefore be discarded. To achieve this, near and far data filters were applied to the data sets which would eliminate anything closer or further than the set values of the near and far filters respectively.

Figure 7: Triangles representing the three-dimensional data.

With near and far depth filters in place, the excess data is removed from the three-dimensional representation which was referred to as the "3-D mesh" at that stage of the project. A significantly cleaner mesh can be seen in Figure 8, where all but the subject of the image is excluded, removing the unwanted objects such as the chairs, tables, and wall that can be seen in the infrared depth image in the upper right hand corner. This stage of the 3-D mesh still contained missing data which caused discontinuities in the mesh which would not allow for the mesh to be a printable object. These "holes" would be addressed at a later stage when a more accurate representation of the mesh could be obtained.

Figure 8: Comparison of depth frame data with background filtered out.

It was determined after initial trials with the software that by not filtering the depth data, the final mesh had a lot of noise which made the figure unpleasant to observe. By implementing a Gaussian-based Bilateral Filter, the results improved significantly and provided an elegant and clean model that could then be printed. This process can best be seen in Figure 9.



(a) Non-filtered scan

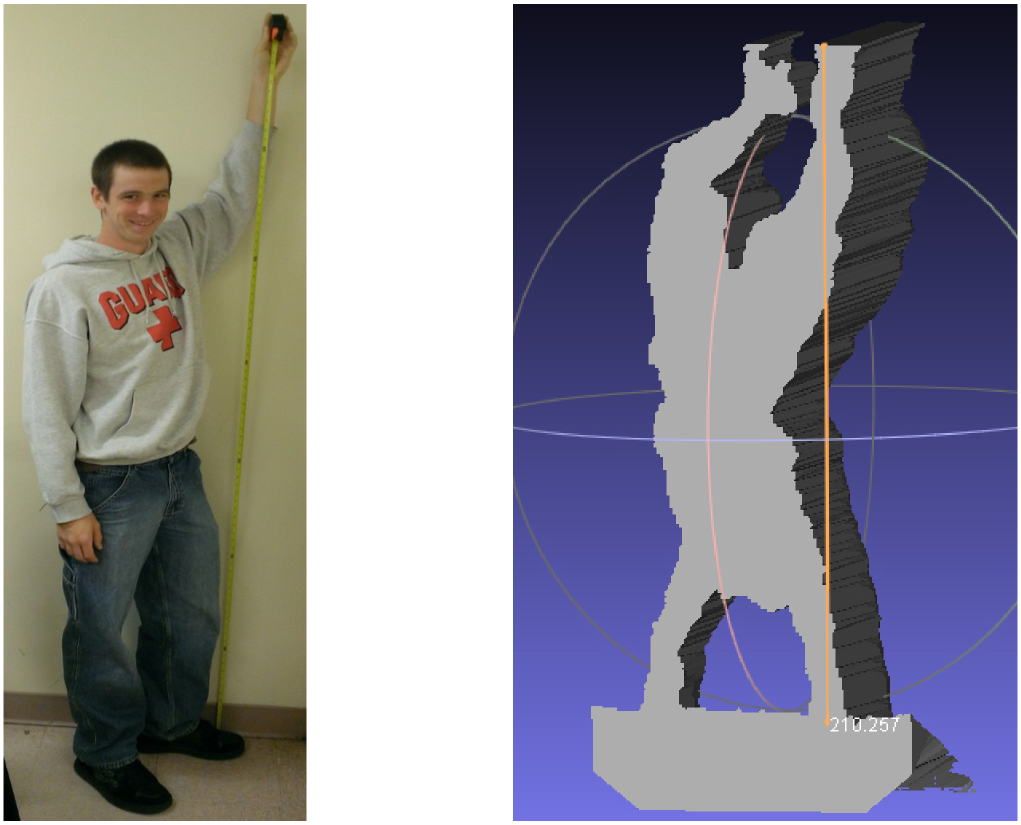
(b) Gaussian filtered scan

Figure 9: As can be seen on the right figure it is clear that filtering the depth data provides a nicer, cleaner, and smoother result.

Through all these trial and error steps a final product of miniature figurines was achieved as can be seen in Figure 10. These figurines were created by scanning the project creators in various poses, then exporting the meshes and creating the machine language G-code which is read by the printer. The final printed object is 20% of the original exported mesh and additionally 100 times smaller from the original depth data, the printed objects are about 50mm tall and on average takes between 20 - 30 minutes to print each one.

Figure 10: Final result - 3-D printed objects that were scanned through the KinectScan application.

To check the accuracy of the three-dimensional models versus real world height, Ryan held a tape measure above his head which came out to approximately 213.36 cm. A height measurement was also performed in the 3-D model software which came out to 210.257 cm. The process can be seen in Figure 11 and comes out to a 1.45% error which shows that the scan is fairly accurate in correctly representing scale and detail.



(a) Actual Height - 7 ft or 213.36 cm (b) 3-D mesh - 6.81 ft or 210.257 cm

Figure 11: Comparison between actual and modeled height, the results are very close.

4 Discussion

An issue with the RepRap 3-D printer is that many of the parts used in the construction of the machine are printed by another 3-D printer. Therefore, we had to wait for all of the correct parts to be printed before we could finish construction of our printer. We discovered that two important parts were missing and therefore we had to go to the Rutgers Maker Space to get these parts printed. The only problem we found with the printed parts was that the holes in the plastic were the exact diameter of the rods that were supposed to fit in. Since forcing the rods into place might have placed too much stress on the plastic parts, we use hot water to make the printed plastic more malleable.

Another issue with the construction of the 3-D printer was the lack of good documentation on the assembly process. The triangular base structure had to be taken apart and reassembled multiple times in order to fix errors. One example was the motor bracket for the motor that controls the movement along the y-axis: there were no diagrams good enough to show which side of the machine the part should be placed and what direction it should be oriented. Figure 12 shows a number of the unlabeled parts that we received from the IEEE. Since documentation was hard to find and none of the parts were labeled, we also had trouble finding the correct STL files to send to get printed. We knew that we were missing the brackets that connect the two top motors to the threaded rod that controls the movement along the z-axis, but we did not know exactly what file needed to be printed.

Figure 12: Parts of the 3-D Printer.

Even after the mechanical parts of the 3-D printer had been constructed, we still needed to calibrate all of the axes, add all of the electrical components, calibrate the firmware, and build the protective frame around the printer. Figure 13 shows the constructed frame, x,y, and z-axes and the installed printer. One of the issues that we were faced with as we completed the 3-D printer was with the extruder. The extruding component is responsible for heating and melting the Acrylonitrile Butadiene Styrene (ABS) plastic and placing it onto the right spot on the heat bed. We discovered a problem in using normal solder since it could not handle the heat that was needed to melt the ABS plastic. As a result, the piece fell apart as the machine heated up. In order to fix the melting issue, we had to order silver solder since it is capable of withstanding 221 degrees Fahrenheit.

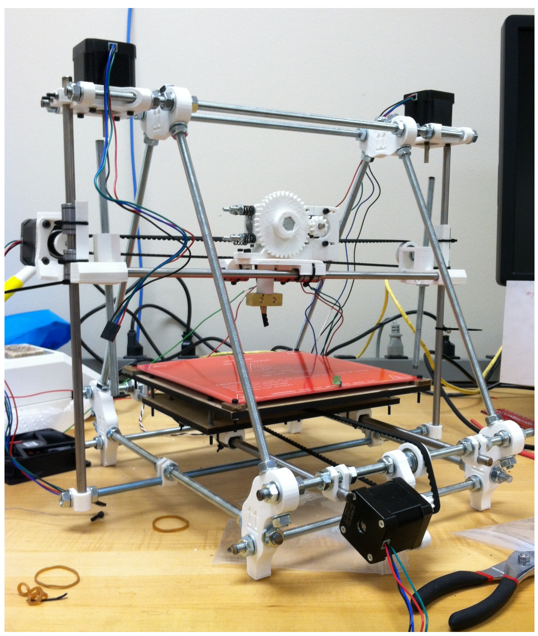


Figure 13: Construction of the 3-D Printer Frame and x,y, and z-axes.

The biggest problem that we experienced with the software component of the project was the lack of examples and demo code to work with. Other people who have worked on similar projects used the original version of the Microsoft Kinect SDK before its official release so many demo implementations required packages that are no longer part of the latest version of the SDK. Instead, we had to find ways to make them work the new SDK, which was released in 2012, and works in a way similar to the old SDK.

Since our application uses the infrared sensor, the scan will not come out at intended under a few conditions. When outside the Kinect cannot collect the data as well as it can inside. At Rutgers Day an all day open house event at Rutgers, we notice that when people stood below the sky light there was essentially no data being collected at the top of their heads because of the sunlight. In addition, our application is dependent on the reflectivity of the object that it is scanning. For example, if someone holds a clear plastic cup in front of their body during the scan, it may appear as if there is a hole in that person's body because the Kinect is unable to record depth data in that area.

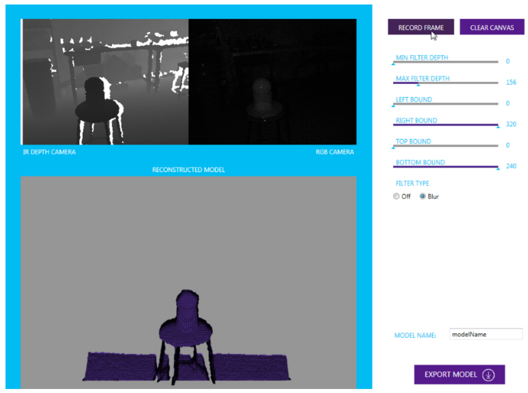


Figure 14: Custom application designed to create three-dimensional meshes - KinectScan

The final version of our custom application, KinecScan, is shown in Figure **??**. The upper left image is a grayscale representation of the depth data that the Kinect is receiving, closer objects are darker in color than objects further away, white indicates that there is no depth data for that pixel. The right image is the RGB camera video feed but set to infrared mode to show the infrared dot matrix. Both of these video streams allow the user to have a good idea of what information will be used to construct the mesh of the current scene. With this application, the user can specify the maximum and minimum x, y and depth values. Adjusting the maximum depth will filter out the background and reconstruct only the object as shown in Figure 14. The x and y filters are useful to crop the window if there is some unmovable object next to the item desired to be scanned, like a column or wall, that is close enough that the depth filer can not filter it out. There is also an option provided to turn the bilateral filter on or off. The two buttons on the top allow the user to record the frame; once this is selected, the three-dimensional reconstructed mesh would appear in the bottom gray window. The button in the upper right corner will clear the canvas and all of the information used to make the three-dimensional mesh. The field at the bottom is used to indicate what the file name will be when the user presses the button at the bottom of the application, that exports the model with the file name specified above it.

5 Cost Analysis

The project in itself is not by any means cost effective in that there is a large cost alone in acquiring a 3-D printer. The printer used in this project costs a little over $550.00 which is not representative of the true expense of large professional 3-D printers; which can cost into the thousands of dollars. Work is currently being performed in industry to lower this cost in order to make it affordable so that even regular consumers may purchase 3-D printers for their homes. [5] The printer however, was payed for through a grant with the IEEE Princeton Central Jersey Section so there was no direct cost to the project.

Other components to the project included a Microsoft Kinect for Xbox which performed 3-D scanning. The cost for this powerful sensor is about$100.00 which for the hardware included is a great price and opens the use of 3-D sensing to just about any project one can think of. The rest of the miscellaneous parts purchased for this project include acrylic casing to make the project more professional. Additional Pololu stepper motor drivers where purchased due to the malfunctioning of the drivers which originally came with the printer kit. A full list of parts used and their costs can be found in Table 1.

Item	Description	Cost
RepRap Prusa Mendel Iteration 2	Open Hardware based 3-D Printer	$ 556.03
Microsoft Kinect for Xbox	Video camera and depth sensor	$ 98.79
Acrylic casing, small tools, glue, misc..	Parts for creating the case	$ 59.65
Pololu A4988 Stepper Motor Driver ($\times 3$)	Converts digital signals to motor movement	$ 33.09
Kinect Power Supply Cable	External power source for Kinect	$ 6.70
	Total Cost:	**$ 754.26**

Table 1: Overview of hardware and cost for project.

6 Current Trends in Robotics and Computer Vision

6.1 Kinect Revolution

One of the reasons that the Kinect has become so popular for computer vision projects is that it is cheap, quick, and highly reliable for three-dimensional measurements. Many researchers are beginning to look into the possibility of using the Kinect to achieve everything from a three-dimensional reconstruction of a scene to aiding in a Simultaneous Localization and Mapping (SLAM) algorithm. The fact that the device is so affordable, and so many new resources are available, makes the Kinect a viable device for conducting research in the field of robotics and computer vision.

The KinectFusion Project is slightly different than other projects that use the Kinect; instead of using both the RGB cameras and the sensor, the project tracks the three-dimensional sensor pose and performs a reconstruction in real time using exclusively the depth data. The KinectFusion paper points out that depth cameras are not exactly new, but the Kinect is a low-cost, real-time, depth camera that is much more accessible. The accuracy of the Kinect is called into question, the point cloud that the depth data creates usually contains noise and sometimes has holes where no readings were obtained. Considering the Kinect's low X/Y resolution and depth accuracy, the project fixes the quality of the images using depth super resolution. KinectFusion looks into using multiple Kinects to perform a three-dimensional body scan; more issues are raised because the quality of the overlapping sections of the images is compromised.

Another KinectFusion Project is the Real-time Three-dimensional Reconstruction and Interaction; it is impressive because the entire process is done using a moving depth camera. With the software, the user can hold a Kinect camera up to a scene, and create a three-dimensional reconstruction in real time. Not only would the user be able to see the three-dimensional reconstruction, but he would be able to interact with it; for instance, if the user were to throw a handful of spheres onto the scene, they would land on the top of appropriate surfaces and fall under appropriate objects following the rules of physics. The depth camera is used to track the three-dimensional pose and the sensor is used to reconstruct the scene in real time. Different views of the scene are taken and fused together into a single representation; the pipe line segments the objects in the scene and uses them to create a global surface based reconstruction. The KinectFusion project shows the real-time capabilities of the Kinect and why it is an innovative tool for computer vision.

A study shown in the Asia Simulation Conference in 2011 demonstrated that a calibrated Kinect can be combined with Structure from Motion to find the three-dimensional data of a scene and reconstruct the surface by multi-view stereo. The study proved that the Kinect was more accurate for this procedure than a SwissRanger SR-4000 three-dimensional-TOF camera and close to a medium resolution SLR Stereo rigs. The Kinect works by using a near-infrared laser pattern projector and an IR camera as a stereo pair to triangulate points in three-dimensional space, then the RGB camera is used to reconstruct the correct texture to the three-dimensional points. The RGB camera, which outputs medium quality images, can also be used for recognition. One issue the study found was that the resulting IR and depth images were shifted. To figure out what the shift was, the Kinect recorded pictures of a circle from different distances. The shift was found to be around 4 pixels in the u direction and three pixels in the v direction. Even after the camera has been fully calibrated, there are a few remaining residual errors in the close range three-dimensional measurements. An easy fix for the error was to form a z-correction image of z values constructed as the pixel-wise mean of all residual images, and then subtract that correction image from the z coordinates of the three-dimensional image.[7] Though the SLR Stereo was the most accurate, the error e (or the Euclidean distance between the points returned by the sensors and points reconstructed in the process of calibration) of the SR-400 was much higher than the Kinect and the SLR. The study shows that the Kinect is a possible cheaper and simpler alternative to previously used cameras and rigs in the computer vision field.

Another subject of research looking into using the Kinect, is the simultaneous localization and mapping algorithm, used to create a three-dimensional map of the world so that the robot can avoid collision with obstacles or walls. The SLAM problem could be solved using GPS if the robot is outside, but while the robot is inside, one needs to use wheel or visual odometry. Visual odometry determines the position and the orientation of the robot using the associated camera images. Algorithms like Scale Invariant Feature Transformation (SIFT), used to find the interest points, and laser sensors, are used to collect depth data. Since the Kinect has both the RGB camera and a laser sensor, the Kinect technology is a good piece of hardware to use for robots computing the SLAM Algorithm. In the study conducted in the Graduates School of Science and Technology at Meiji University, the students found that the Kinect worked well for the SLAM process for horizontal and straight movement, but they had errors when they tried to recreate an earlier experiment. Their algorithm successfully solves the initial problem, but accuracy fell over time.[10] The students found that the issue was not with the Kinect, and that it could be solved using the Speed-Up Robust Feature algorithm (SURF) and Smirnov-Grubbs test to further improve the accuracy of their SLAM Algorithm. The study proved that the Kinect was a reasonable, inexpensive and non-special piece of equipment that is capable of performing well in computer vision applications.

It seems as though the Kinect is a popular choice of camera and depth sensor in current robotics and computer vision. The Kinect device is affordable, easily obtainable, and capable of a lot more than is expected from a video game add on. The Kinect is surprisingly accurate, requiring minimal calibration and only some optimization software to make the results comparable to the results from a medium resolution SLR Stereo rig.

6.2 3-D Printing Future

One of the most innovative uses for the 3-D printer is its application in the medical field. Since 2010, people have been using 3-D printers to print out prosthetic limbs. One company in California has been printing the customizable prosthetics, which cost about one tenth of traditional prosthetics limbs. Another company is looking at the possibility of using a 3-D printer to print a house. Currently, the design fits on the back of a tractor trailer and the 3-D printer prints out custom concrete parts that are assembled to complete the house. Some 3-D printers have the ability to change the printing head, so it can begin printing with one material and then switch to a different material, all based on the code it receives. That a 3-D printer could theoretically print the concrete part of the house and switch to printing the plastic siding or the glass windows, all on the same path around the outside of the house. The most important aspect of these 3-D printer's application is that it drastically cuts down on production costs, allowing the consumer to pay a lower price and get a completely customized product. Rather than paying a person to design the object and have a another construct it, with a 3-D printer all that needs to be done is the design and the 3-D printer automates the entire construction process. For example, the three-dimensional printed prosthetics cost 5,000 dollars to print and customize by covering the three-dimensional printed material in a shoe or sleeve while a normal generic prosthetics would cost about 60,000 dollars.[11] The 3-D printer is a piece of technology that could continue to make the price of consumer goods fall and allow for more customization than has ever existed for consumer products.

In recent news, biomedical scientists have taken the three-dimensional printing technology a step further than prosthetics. A man had 75% of his skull replaced but a three-dimensional printed implant made by Oxford Performance Materials. since the 1940's, normal plastics have been used to replace missing bone fragments; now, three-dimensional modeling techniques can be used to exactly match the size and shape of the plastic to someones skull. The Connecticut based company combined the three-dimensional modeling techniques with three-dimensional printing technology to produce the replacement part that took only five days to fabricate.[5] The material used has some of the same properties as bones and are osteoconductive, meaning the skull will actually grow and attach itself to the implant. The plastic is much better than metals, which would block doctors from seeing past the implant in X-rays. The company is now also looking at using this procedure to three-dimensional print other replacement bones for victims of cancerous bone or trauma.

Even though lower cost of production is a goal for many industries, the three-dimensional printing technology can be considered a disruptive technology, meaning that over the course of a short period of time it could change an existing market and value network, while replacing existing technology. An article in the Harvard Business Review explains that goods would be produced at or close to their point of purchase or consumption. Even if this is not the case with every industry, the cost will be offset by the elimination of shipping of the completed object to the consumer, something like car parts could be printed in a metropolitan area rather than made and shipped from a factory. The article also mentions how the 3-D printer would allow for cheap and efficient customization of these products. Since changing the shape, color, or material of what the machine is printing is only a mater of changing code, the first model could be relatively different than the second model for virtually no extra cost. [8] The 3-D printer could also potentially affect the global market. Many products are manufactured overseas since it is much cheaper for the pieces to be created and assembled by underpaid workers. When three-dimensional printing is perfected, the parts could be made and assembled by a machine in the US for less than it costs to have the product manufactured and shipped from overseas.

An article in Machine Design talks about what changes are being made to the 3-D printers in order to make them more durable, user friendly, and affordable. One brand, LeapFrog, has made the entire device out of aluminum and replaced the stepper-motor drivers with professional drivers that last longer. The company has also added a dual option extruder so that the printer could construct something like a bridge by adding the plastic from one extruder and a water soluble support system with the other extrude. The water soluble support system can be easily washed away once the printing is finished. The printer also uses PLA plastic, which is more brittle and has a lower melting temperature and can print smoother edges than the usual ABS plastic. Another brand, FormLabs, uses a liquid photopolymer instead of a spool of plastic. The resin cuts the price of printing materials in half and allows for a layer thickness of only 25 microns. The RepRap 3-D printer has been designated a self-replicating printer because it can be used to print parts for constructing another 3-D printer. It is believed that between 20,000 and 30,000 of these machines are now in existence.[9] The company Staples has started "Staples Easy three-dimensional" in Belgium and the Netherlands, where anyone can upload their file to the center and later pick up the three-dimensional model at their local Staples or have it shipped to their house. Services like this are a sign that three-dimensional printing will soon be as mainstream as two-dimensional printing is.

7 Acknowledgment

This project was made possible by a research grant sponsored by the IEEE Princeton Central Jersey Section, it is with their support of the donation of the RepRap Prusa Mendel Iteration 2 3-D Printer and a grant of $250 that the project was able to be successfully completed.

We would also like to thank the Rutgers Department of Electrical & Computer Engineering, Dr. Kristin Dana, Rebecca Mercuri, Kevin Meredith, John Scafidi, Steve Orbine, and Michael DiLalo for supporting and assisting with our work.

8 Appendix

8.1 KinectScan Application Code

```
ïż£namespace kinectScan
{
    using System;
    using System.ComponentModel;
    using System.Globalization;
    using System.IO;
    using System.Threading.Tasks;
    using System.Drawing;
    using System.Diagnostics;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Media;
    using System.Windows.Media.Imaging;
    using System.Windows.Media.Media3D;
    using System.Windows.Threading;

    using HelixToolkit.Wpf;

    using Microsoft.Kinect;
    using Microsoft.Kinect.Toolkit;

    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {

        /// <summary>
```

```
        /// Timestamp of last depth frame in milliseconds
        /// </summary>
        private long lastFrameTimestamp = 0;

        /// <summary>
        /// Timer to count FPS
        /// </summary>
        private DispatcherTimer fpsTimer;

        /// <summary>
        /// Timer stamp of last computation of FPS
        /// </summary>
        private DateTime lastFPSTimestamp;

        /// <summary>
        /// Event interval for FPS timer
        /// </summary>
        private const int FpsInterval = 5;

        /// <summary>
        /// The counter for frames that have been processed
        /// </summary>
        private int processedFrameCount = 0;

        /// <summary>
        /// Active Kinect sensor
        /// </summary>
        private KinectSensor sensor;

        /// <summary>
        /// Kinect sensor chooser object
        /// </summary>
        private KinectSensorChooser sensorChooser;

        /// <summary>
        /// Format of depth image to use
        /// </summary>
        private const DepthImageFormat dFormat = DepthImageFormat.Resolution320x240Fps30;

        /// <summary>
        /// Format of color image to use
        /// </summary>
```

```
        private const ColorImageFormat cFormat = ColorImageFormat.
            InfraredResolution640x480Fps30;

        // stores furthest depth in the scene
        public ushort greatestDepth = 0;

        // array for all of the depth data
        private int[] Depth = new int[320 * 240];

        // stores all of the 3D trianlges with normals and points
        Model3DGroup modelGroup = new Model3DGroup();

        // material placed over the mesh for viewing
        public GeometryModel3D msheet = new GeometryModel3D();

        // collection of corners for the triangles
        public Point3DCollection corners = new Point3DCollection();

        // collection of all the triangles
        public Int32Collection Triangles = new Int32Collection();


        public MeshGeometry3D tmesh = new MeshGeometry3D();

        // collection of all the cross product normals
        public Vector3DCollection Normals = new Vector3DCollection();

        // add texture to the mesh
        public PointCollection myTextureCoordinatesCollection = new PointCollection();

        // storage for camera, scene, etc...
        public ModelVisual3D modelsVisual = new ModelVisual3D();


        public Viewport3D myViewport = new Viewport3D();

        // test variable
        public int samplespot;

        // variable for changing the quality 1 is the best 16 contains almost no data
        public int s = 1;

        // depth point collection
```

```
        public int[] depths_array = new int[4];

        // collection of points
        Point3D[] points_array = new Point3D[4];

        // collection of vectors
        Vector3D[] vectors_array = new Vector3D[5];

        //used for displaying RGB camera
        public byte[] colorPixels;
        public WriteableBitmap colorBitmap;

        public MainWindow()
        {
            InitializeComponent();
        }

        private void WindowLoaded(object sender, RoutedEventArgs e)
        {
            // Start Kinect sensor chooser
            this.sensorChooser = new KinectSensorChooser();
            this.sensorChooserUI.KinectSensorChooser = this.sensorChooser;
            this.sensorChooser.KinectChanged += this.OnKinectSensorChanged;
            this.sensorChooser.Start();

            // Start fps timer
            this.fpsTimer = new DispatcherTimer(DispatcherPriority.Send);
            this.fpsTimer.Interval = new TimeSpan(0, 0, FpsInterval);
            this.fpsTimer.Tick += this.FpsTimerTick;
            this.fpsTimer.Start();

            // Set last fps timestamp as now
            this.lastFPSTimestamp = DateTime.Now;
        }

        /// <summary>
        /// Execute shutdown tasks
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void WindowClosing(object sender, System.ComponentModel.CancelEventArgs e
            )
        {
```

```
            // Stop timer
            if (null != this.fpsTimer)
            {
                this.fpsTimer.Stop();
                this.fpsTimer.Tick -= this.FpsTimerTick;
            }

            // Unregister Kinect sensor chooser event
            if (null != this.sensorChooser)
            {
                this.sensorChooser.KinectChanged -= this.OnKinectSensorChanged;
            }

            // Stop sensor
            if (null != this.sensor)
            {
                this.sensor.Stop();
                this.sensor.DepthFrameReady -= this.SensorDepthFrameReady;
                this.sensor.ColorFrameReady -= this.SensorColorFrameReady;
            }

            // Empty the canvas
            this.ClearMesh();
        }

        /// <summary>
        /// Handles adding a new kinect
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments for the newly connected Kinect</param>
        private void OnKinectSensorChanged(object sender, KinectChangedEventArgs e)
        {
            // Check new sensor's status
            if (this.sensor != e.NewSensor)
            {
                // Stop old sensor
                if (null != this.sensor)
                {
                    this.sensor.Stop();
                    this.sensor.DepthFrameReady -= this.SensorDepthFrameReady;
                    this.sensor.ColorFrameReady -= this.SensorColorFrameReady;
                }

```

```
                this.sensor = null;

                if (null != e.NewSensor && KinectStatus.Connected == e.NewSensor.Status)
                {
                    // Start new sensor
                    this.sensor = e.NewSensor;
                    this.StartCameraStream(dFormat, cFormat);
                }
            }

            if (null == this.sensor)
            {
                // if no kinect clear the text on screen
                this.statusBarText.Content = Properties.Resources.NoKinectReady;
                this.IR_Title.Content = "";
                this.Model_Title.Content = "";
                this.RGB_Title.Content = "";
            }
        }

        /// <summary>
        /// Handler for FPS timer tick
        /// </summary>
        /// <param name="sender">Object sending the event</param>
        /// <param name="e">Event arguments</param>
        private void FpsTimerTick(object sender, EventArgs e)
        {

            if (null == this.sensor)
            {
                // Show "No ready Kinect found!" on status bar
                this.KinectStatusText.Content = Properties.Resources.NoReadyKinect;
            }
            else
            {
                // Calculate time span from last calculation of FPS
                double intervalSeconds = (DateTime.Now - this.lastFPSTimestamp).
                    TotalSeconds;

                // Calculate and show fps on status bar
                this.KinectStatusText.Content = string.Format(
                    System.Globalization.CultureInfo.InvariantCulture,
                    Properties.Resources.Fps,
```

```
                    (double) this.processedFrameCount / intervalSeconds);
            }

            // Reset frame counter
            this.processedFrameCount = 0;
            this.lastFPSTimestamp = DateTime.Now;
        }

        /// <summary>
        /// Reset FPS timer and counter
        /// </summary>
        private void ResetFps()
        {
            // Restart fps timer
            if (null != this.fpsTimer)
            {
                this.fpsTimer.Stop();
                this.fpsTimer.Start();
            }

            // Reset frame counter
            this.processedFrameCount = 0;
            this.lastFPSTimestamp = DateTime.Now;
        }

        /// <summary>
        /// Start depth stream at specific resolution
        /// </summary>
        /// <param name="format">The resolution of image in depth stream</param>
        private void StartCameraStream(DepthImageFormat dFormat, ColorImageFormat cFormat
            )
        {
            try
            {
                // Enable streams, register event handler and start
                this.sensor.DepthStream.Enable(dFormat);
                this.sensor.DepthFrameReady += this.SensorDepthFrameReady;
                this.sensor.ColorStream.Enable(cFormat);
                this.sensor.ColorFrameReady += this.SensorColorFrameReady;
                this.sensor.Start();
            }
            catch (IOException ex)
            {
```

```
                // Device is in use
                this.sensor = null;
                this.ShowStatusMessage(ex.Message);

                return;
            }
            catch (InvalidOperationException ex)
            {
                // Device is not valid, not supported or hardware feature unavailable
                this.sensor = null;
                this.ShowStatusMessage(ex.Message);

                return;
            }

            // Allocate space to put the pixels we'll receive
            this.colorPixels = new byte[this.sensor.ColorStream.FramePixelDataLength];

            //// This is the bitmap we'll display on-screen
            this.colorBitmap = new WriteableBitmap(this.sensor.ColorStream.FrameWidth,
                this.sensor.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Gray16, null
                );
        }

        /// <summary>
        /// Event handler for Kinect sensor's ColorFrameReady event
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        void SensorColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
        {
            using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
            {
                if (colorFrame != null)
                {
                    // Copy the pixel data from the image to a temporary array
                    colorFrame.CopyPixelDataTo(this.colorPixels);

                    // Write the pixel data into our bitmap
                    this.colorBitmap.WritePixels(
                        new Int32Rect(0, 0, this.colorBitmap.PixelWidth, this.colorBitmap
                            .PixelHeight),
                        this.colorPixels,
```

```
                    this.colorBitmap.PixelWidth * colorFrame.BytesPerPixel,
                    0);
            }

            // set the RGB image to the RGB camera
            this.KinectRGBView.Source = this.colorBitmap;

        }
    }

    /// <summary>
    /// Event handler for Kinect sensor's DepthFrameReady event
    /// Take in depth data
    /// </summary>
    /// <param name="sender">object sending the event</param>
    /// <param name="e">event arguments</param>
    void SensorDepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
    {

        DepthImageFrame imageFrame = e.OpenDepthImageFrame();
        if (imageFrame != null)
        {
            double maxDepth = Far_Filter_Slider.Value;
            short[] pixelData = new short[imageFrame.PixelDataLength];
            imageFrame.CopyPixelDataTo(pixelData);
            this.greatestDepth = 0;
            for (int y = 0; y < 240; y++)
            {
                for (int x = 0; x < 320; x++)
                {
                    // scale depth down
                    this.Depth[x + (y * 320)] = ((ushort)pixelData[x + y * 320]) /
                        100;

                    // finds the furthest depth from all the depth pixels
                    if ((this.Depth[x + y * 320] > this.greatestDepth) && (this.Depth
                        [x + y * 320] < maxDepth))
                    {
                        this.greatestDepth = (ushort)this.Depth[x + y * 320];
                    }


                }
```

```
                }
                // Blur Filter -- Guassian
                if (Filter_Blur.IsChecked == true)
                {
                    for (int i = 641; i < this.Depth.Length - 641; ++i)
                    {

                        short depthaverage = (Int16)((this.Depth[i - 641] + (2 * this.
                            Depth[i - 640]) + this.Depth[i - 639] +
                                                              (2 * this.Depth[i - 1]) + (4 * this.
                                                                  Depth[i]) + (2 * this.Depth[i +
                                                                  2]) +
                                                              this.Depth[i + 639] + (2 * this.
                                                                  Depth[i + 640]) + this.Depth[i +
                                                                  641]) / 16);

                        this.Depth[i] = depthaverage;
                        if ((this.Depth[i] > this.greatestDepth) && (this.Depth[i] <
                            maxDepth))
                        {
                            this.greatestDepth = (ushort)this.Depth[i];
                        }
                    }
                }

                // Set the depth image to the Depth sensor view
                this.KinectDepthView.Source = DepthToBitmapSource(imageFrame);
            }
        }


        /// <summary>
        /// Flag check for a point within the bounding box
        /// </summary>
        /// <param name="x">location on the x plane</param>
        /// <param name="y">location on the y plane</param>
        private bool PointinRange(int x, int y)
        {
            double minDepth = Near_Filter_Slider.Value;
            double maxDepth = Far_Filter_Slider.Value;
            return ((this.Depth[x + (y * 320)] >= minDepth && this.Depth[x + (y * 320)]
                <= maxDepth) ||
```

```
            (this.Depth[(x + s) + (y * 320)] >= minDepth && this.Depth[(x + s) + (y *
                320)] <= maxDepth) ||
            (this.Depth[x + ((y + s) * 320)] >= minDepth && this.Depth[x + ((y + s) *
                320)] <= maxDepth) ||
            (this.Depth[(x + s) + ((y + s) * 320)] >= minDepth && this.Depth[(x + s)
                + ((y + s) * 320)] <= maxDepth));

    }

    /// <summary>
    /// Create the mesh
    /// </summary>
    void BuildMesh()
    {
        double maxDepth = Far_Filter_Slider.Value;
        int i = 0;
        for (int y = (int)Top_Slider.Value; y < ((int)Bot_Slider.Value - s); y = y +
            s)
        {
            for (int x = (int)Left_Slider.Value; x < ((int)Right_Slider.Value - s); x
                = x + s)
            {
                //Any point less than max
                if (PointinRange(x, y))
                {
                    if (this.Depth[x + ((y + s) * 320)] >= maxDepth)
                    {
                        depths_array[0] = -this.greatestDepth;
                    }
                    else
                    {
                        depths_array[0] = -this.Depth[x + ((y + s) * 320)];
                    }

                    if (this.Depth[x + (y * 320)] >= maxDepth)
                    {
                        depths_array[1] = -this.greatestDepth;
                    }
                    else
                    {
                        depths_array[1] = -this.Depth[x + (y * 320)];
                    }

```

```
                    if (this.Depth[(x + s) + (y * 320)] >= maxDepth)
                    {
                        depths_array[2] = -this.greatestDepth;
                    }
                    else
                    {
                        depths_array[2] = -this.Depth[(x + s) + (y * 320)];
                    }

                    if (this.Depth[(x + s) + ((y + s) * 320)] >= maxDepth)
                    {
                        depths_array[3] = -this.greatestDepth;
                    }
                    else
                    {
                        depths_array[3] = -this.Depth[(x + s) + ((y + s) * 320)];
                    }

                    // triangle point locations
                    points_array[0] = new Point3D(x, (y + s), depths_array[0]);
                    points_array[1] = new Point3D(x, y, depths_array[1]);
                    points_array[2] = new Point3D((x + s), y, depths_array[2]);
                    points_array[3] = new Point3D((x + s), (y + s), depths_array[3]);

                    // create vectors of size difference between points
                    vectors_array[0] = new Vector3D(points_array[1].X - points_array
                        [0].X, points_array[1].Y - points_array[0].Y, points_array[1].
                        Z - points_array[0].Z);
                    vectors_array[1] = new Vector3D(points_array[1].X - points_array
                        [2].X, points_array[1].Y - points_array[2].Y, points_array[1].
                        Z - points_array[2].Z);
                    vectors_array[2] = new Vector3D(points_array[2].X - points_array
                        [0].X, points_array[2].Y - points_array[0].Y, points_array[2].
                        Z - points_array[0].Z);
                    vectors_array[3] = new Vector3D(points_array[3].X - points_array
                        [0].X, points_array[3].Y - points_array[0].Y, points_array[3].
                        Z - points_array[0].Z);
                    vectors_array[4] = new Vector3D(points_array[2].X - points_array
                        [3].X, points_array[2].Y - points_array[3].Y, points_array[2].
                        Z - points_array[3].Z);

                    // add the corners to the 2 triangles to form a square
                    corners.Add(points_array[0]);
```

```
                corners.Add(points_array[1]);
                corners.Add(points_array[2]);
                corners.Add(points_array[2]);
                corners.Add(points_array[3]);
                corners.Add(points_array[0]);

                // add triangles to the collection
                Triangles.Add(i);
                Triangles.Add(i + 1);
                Triangles.Add(i + 2);
                Triangles.Add(i + 3);
                Triangles.Add(i + 4);
                Triangles.Add(i + 5);

                // find the normals of the triangles by taking the cross product
                Normals.Add(Vector3D.CrossProduct(vectors_array[0], vectors_array
                    [2]));
                Normals.Add(Vector3D.CrossProduct(vectors_array[0], vectors_array
                    [1]));
                Normals.Add(Vector3D.CrossProduct(vectors_array[1], vectors_array
                    [2]));
                Normals.Add(Vector3D.CrossProduct(vectors_array[1], vectors_array
                    [2]));
                Normals.Add(Vector3D.CrossProduct(vectors_array[3], vectors_array
                    [4]));
                Normals.Add(Vector3D.CrossProduct(vectors_array[0], vectors_array
                    [2]));

                i = i + 6;
            }

        }
    }

    // add the flat back wall
    int numcorners = corners.Count;
    for (int p = 0; p < numcorners; p++)
    {
        Point3D cornertocopy = corners[p];
        corners.Add(new Point3D(cornertocopy.X, cornertocopy.Y, -this.
            greatestDepth));
        Triangles.Add(i);
        Normals.Add(new Vector3D(0, 0, 1));
```

```
                i = i + 1;
            }


        }

        /// <summary>
        /// Create depth image from depth frame
        /// </summary>
        /// <param name="imageFrame">collection of depth data</param>
        BitmapSource DepthToBitmapSource(DepthImageFrame imageFrame)
        {
            short[] pixelData = new short[imageFrame.PixelDataLength];
            imageFrame.CopyPixelDataTo(pixelData);
            BitmapSource bmap = BitmapSource.Create(
             imageFrame.Width,
             imageFrame.Height,
             96, 96,
             PixelFormats.Gray16,
             null,
             pixelData,
             imageFrame.Width * imageFrame.BytesPerPixel);
            return bmap;
        }

        /// <summary>
        /// take a photo when button is clicked
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Begin_Scan_Click(object sender, RoutedEventArgs e)
        {
            //clear the canvas
            this.ClearMesh();

            // add light to the scene
            DirectionalLight DirLight1 = new DirectionalLight();
            DirLight1.Color = Colors.White;
            DirLight1.Direction = new Vector3D(0, 0, -1);

            // add a camera to the scene
            PerspectiveCamera Camera1 = new PerspectiveCamera();

```

```
            // set the location of the camera
            Camera1.Position = new Point3D(160, 120, 480);
            Camera1.LookDirection = new Vector3D(0, 0, -1);
            Camera1.UpDirection = new Vector3D(0, -1, 0);

            // create the mesh from depth data
            this.BuildMesh();

            // add texture to all the points
            tmesh.Positions = corners;
            tmesh.TriangleIndices = Triangles;
            tmesh.Normals = Normals;
            tmesh.TextureCoordinates = myTextureCoordinatesCollection;
            msheet.Geometry = tmesh;
            msheet.Material = new DiffuseMaterial((SolidColorBrush)(new BrushConverter().
                ConvertFrom("#52318F")));

            // build the scene and display it
            this.modelGroup.Children.Add(msheet);
            this.modelGroup.Children.Add(DirLight1);
            this.modelsVisual.Content = this.modelGroup;
            this.myViewport.IsHitTestVisible = false;
            this.myViewport.Camera = Camera1;
            this.myViewport.Children.Add(this.modelsVisual);
            KinectNormalView.Children.Add(this.myViewport);
            this.myViewport.Height = KinectNormalView.Height;
            this.myViewport.Width = KinectNormalView.Width;
            Canvas.SetTop(this.myViewport, 0);
            Canvas.SetLeft(this.myViewport, 0);

        }

        /// <summary>
        /// Export the completed mesh to a .obj file
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void Export_Model_Click(object sender, RoutedEventArgs e)
        {
            //function from Helix Toolkit
            string fileName = Model_Name.Text + ".obj";

            using (var exporter = new ObjExporter(fileName))
```

```
            {
                exporter.Export(this.modelGroup);
            }

            // test code for seeing depth frame values
            Process.Start("explorer.exe", "/select,\"" + fileName + "\"");

            string fileName2 = "depth.txt";

            using (System.IO.StreamWriter file = new System.IO.StreamWriter(fileName2))
            {
                //file.Write(string.Join(",", this.Depth));
                file.Write(greatestDepth);
            }

        }

        /// <summary>
        /// Show exception info on status bar
        /// </summary>
        /// <param name="message">Message to show on status bar</param>
        private void ShowStatusMessage(string message)
        {
            this.Dispatcher.BeginInvoke((Action)(() =>
            {
                this.ResetFps();
                this.KinectStatusText.Content = message;
            }));
        }

        /// <summary>
        /// clear everything from the scene and canvas
        /// </summary>
        public void ClearMesh()
        {
            KinectNormalView.Children.Clear();
            modelGroup.Children.Clear();
            myViewport.Children.Clear();
            modelsVisual.Children.Clear();
            tmesh.Positions.Clear();
            tmesh.TriangleIndices.Clear();
            tmesh.Normals.Clear();
            tmesh.TextureCoordinates.Clear();
```

```
        }

        /// <summary>
        /// Clear canvas button click
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        private void End_Scan_Click(object sender, RoutedEventArgs e)
        {
            this.ClearMesh();
        }
    }
}
```

8.2 KinectScan Graphical User Interface Code

```
ïż£<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:kinectScan"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" mc:Ignorable="d" x:Class="kinectScan.MainWindow"
        xmlns:tk="clr-namespace:Microsoft.Kinect.Toolkit;assembly=Microsoft.Kinect.Toolkit"
        Title="kinectScan" Height="870" Width="1028" Loaded="WindowLoaded" Closing="WindowClosing" Top="0" Left="0" Icon="Images/Kinect.ico">

    <Window.Resources>

        <ResourceDictionary Source="/KinectResources.xaml" />

    </Window.Resources>

    <Grid x:Name="LayoutGrid"  Margin="0,␣0,␣0,␣0">

        <Grid.RowDefinitions>
            <RowDefinition />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition Width="700" />
            <ColumnDefinition Width="30" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Rectangle Fill="{StaticResource␣SecondaryBrandBrush}" />

        <Grid x:Name="CameraZone" Margin="0,0,0,0" TextBlock.FontFamily="{StaticResource␣
            KinectFont}" Grid.Column="0">

            <Grid.RowDefinitions>
                <RowDefinition Height="270" />
                <RowDefinition Height="30"/>
                <RowDefinition Height="30"/>
                <RowDefinition Height="510" />
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="700" />
            </Grid.ColumnDefinitions>

            <!-- Depth Camera -->
            <Rectangle Fill="{StaticResource␣MediumNeutralBrush}" Grid.Row="0" Height="
                240" Width="320" Margin="30,30,350,0" />
            <Image Name="KinectDepthView" Grid.Row="0" Height="240" Width="320" Margin="
                30,30,350,␣0"/>

            <!-- Bilateral Camera-->
            <Rectangle Fill="{StaticResource␣MediumNeutralBrush}" Grid.Row="0" Height="
                240" Width="320" Margin="350,30,30,0" />
            <Image Name="KinectRGBView" Grid.Row="0" Height="240" Width="320" Margin="
                350,30,30,0"/>

            <!-- Reconstruction Model -->
            <Grid x:Name="Reconstruction_Grid" Grid.Row="3">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="30"/>
                    <ColumnDefinition/>
                    <ColumnDefinition Width="30"/>
                </Grid.ColumnDefinitions>

                <Grid.RowDefinitions>
                    <RowDefinition Height="480"/>
```

```
                <RowDefinition/>
            </Grid.RowDefinitions>

            <Rectangle Fill="{StaticResource MediumNeutralBrush}" Grid.Row="3" Height="480" Width="640" Margin="30,0,30,30" />
            <Canvas Name="KinectNormalView" Grid.Column="1" Height="480" Width="640" Margin="0,0,0,30" Background="{StaticResource MediumNeutralBrush}"/>

            <!-- Bounding Box-->
            <!-- <Border BorderBrush="Red" BorderThickness="1" Grid.Column="1" /> -->

        </Grid>

        <!-- Titles -->
        <Label x:Name="IR_Title" Content="IR DEPTH CAMERA"  Grid.Row="1" Foreground="White" HorizontalAlignment="Left" Margin="30,0,0,0" VerticalAlignment="Top"/>
        <Label x:Name="RGB_Title" Content="RGB CAMERA"  Grid.Row="1" Foreground="White" HorizontalAlignment="Right" Margin="0,0,30,0" VerticalAlignment="Top"/>
        <Label x:Name="Model_Title" Content="RECONSTRUCTED MODEL" Grid.Row="2" Foreground="White" HorizontalAlignment="Center" Margin="0,0,0,0" VerticalAlignment="Bottom"/>
        <Label x:Name="statusBarText" Grid.Row="1" Foreground="White" HorizontalAlignment="Center" Margin="0,0,0,0" VerticalAlignment="Center" Grid.RowSpan="2"/>
        <Label x:Name="KinectStatusText" Content="Kinect Status: Loading..."  Grid.Row="3" Foreground="White" HorizontalAlignment="Left" Margin="10,0,0,5" VerticalAlignment="Bottom"/>

    </Grid>
    <!--CameraZone-->

    <Grid x:Name="MenuArea" Background="White" Grid.Column="2">

        <Grid.RowDefinitions>
            <RowDefinition Height="90" />
            <RowDefinition Height="240" />
            <RowDefinition Height="50" />
            <RowDefinition />
            <RowDefinition Height="30" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
```

```

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="290" />
        </Grid.ColumnDefinitions>

        <Button x:Name ="Begin_Scan" Content="RECORD␣FRAME" Margin="0,30,0,0" Style="
            {StaticResource␣KinectButton}" Grid.Row="0" Click="Begin_Scan_Click"/>
        <Button x:Name ="End_Scan" Content="CLEAR␣CANVAS" Margin="137,30,0,0" Style="
            {StaticResource␣KinectButton}" Grid.Row="0" Click="End_Scan_Click" />

        <!--BeginSlider Area-->
        <Grid x:Name="SliderArea" Background="White" Grid.Row="1" Margin="0,0,30,30"
            Grid.RowSpan="2">
            <Grid.RowDefinitions>
                <RowDefinition Height="40" />
                <RowDefinition Height="40" />
                <RowDefinition Height="40" />
                <RowDefinition Height="40" />
                <RowDefinition Height="40" />
                <RowDefinition Height="40" />
                <RowDefinition Height="40" />
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="220" />
                <ColumnDefinition Width="40" />
            </Grid.ColumnDefinitions>

            <Label x:Name="Near_Filter_Title" Content="MIN␣FILTER␣DEPTH" Foreground="
                {StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left"
                Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="0" Grid.Column="0"
                 />
            <Slider x:Name="Near_Filter_Slider" HorizontalAlignment="Left"  Margin="
                10,20,0,0" VerticalAlignment="Top" Width="200" Style="{StaticResource␣
                SliderStyle}" Grid.Row="0" Grid.Column="0" Minimum="0" Maximum="654"
                Value="0"/>
            <Label x:Name="Near_Filter_Value" Content="{Binding␣ElementName=
                Near_Filter_Slider,Path=Value}" ContentStringFormat="{}{0:N0}" Grid.
                Row="0" Grid.Column="1" Foreground="{StaticResource␣
                SecondaryBrandBrush}" HorizontalAlignment="Left" VerticalAlignment="
                Center" />

```

```
<Label x:Name="Far_Filter_Title" Content="MAX␣FILTER␣DEPTH" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="1" Grid.Column="0" />
<Slider x:Name="Far_Filter_Slider" HorizontalAlignment="Left"  Margin="10,20,0,0" VerticalAlignment="Top" Width="200" Style="{StaticResource␣SliderStyle}" Grid.Row="1" Grid.Column="0" Minimum="{Binding␣ElementName=Near_Filter_Slider,Path=Value}" Maximum="654" Value="300"/>
<Label x:Name="Far_Filter_Value" Content="{Binding␣ElementName=Far_Filter_Slider,Path=Value}" ContentStringFormat="{}{0:N0}" Grid.Row="1" Grid.Column="1" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" VerticalAlignment="Center" />

<Label x:Name="Left_Title" Content="LEFT␣BOUND" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="2" Grid.Column="0" />
<Slider x:Name="Left_Slider" HorizontalAlignment="Left"  Margin="10,20,0,0" VerticalAlignment="Top" Width="200" Style="{StaticResource␣SliderStyle}" Grid.Row="2" Grid.Column="0" Minimum="0" Maximum="320" Value="0"/>
<Label x:Name="Left_Value" Content="{Binding␣ElementName=Left_Slider,Path=Value}" ContentStringFormat="{}{0:N0}" Grid.Row="2" Grid.Column="1" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" VerticalAlignment="Center" />

<Label x:Name="Right_Title" Content="RIGHT␣BOUND" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="3" Grid.Column="0" />
<Slider x:Name="Right_Slider" HorizontalAlignment="Left"  Margin="10,20,0,0" VerticalAlignment="Top" Width="200" Style="{StaticResource␣SliderStyle}" Grid.Row="3" Grid.Column="0" Minimum="0" Maximum="320" Value="320"/>
<Label x:Name="Right_Value" Content="{Binding␣ElementName=Right_Slider,Path=Value}" ContentStringFormat="{}{0:N0}" Grid.Row="3" Grid.Column="1" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" VerticalAlignment="Center" />

<Label x:Name="Top_Title" Content="TOP␣BOUND" Foreground="{StaticResource␣SecondaryBrandBrush}" HorizontalAlignment="Left" Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="4" Grid.Column="0" />
<Slider x:Name="Top_Slider" HorizontalAlignment="Left"  Margin="10,20,0,0" VerticalAlignment="Top" Width="200" Style="{StaticResource␣SliderStyle}" Grid.Row="4" Grid.Column="0" Minimum="0" Maximum="240"
```

```
            Value="0"/>
        <Label x:Name="Top_Value" Content="{Binding ElementName=Top_Slider,Path=Value}" ContentStringFormat="{}{0:N0}" Grid.Row="4" Grid.Column="1" Foreground="{StaticResource SecondaryBrandBrush}" HorizontalAlignment="Left" VerticalAlignment="Center" />

        <Label x:Name="Bot_Title" Content="BOTTOM BOUND" Foreground="{StaticResource SecondaryBrandBrush}" HorizontalAlignment="Left" Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="5" Grid.Column="0" />
        <Slider x:Name="Bot_Slider" HorizontalAlignment="Left"  Margin="10,20,0,0" VerticalAlignment="Top" Width="200" Style="{StaticResource SliderStyle}" Grid.Row="5" Grid.Column="0" Minimum="0" Maximum="240" Value="240"/>
        <Label x:Name="Bot_Value" Content="{Binding ElementName=Bot_Slider,Path=Value}" ContentStringFormat="{}{0:N0}" Grid.Row="5" Grid.Column="1" Foreground="{StaticResource SecondaryBrandBrush}" HorizontalAlignment="Left" VerticalAlignment="Center" />
    </Grid>
    <!--EndSliderArea-->

    <!--Begin Radio-->
    <Label x:Name="Filter_Type_Title" Content="FILTER TYPE" Foreground="{StaticResource SecondaryBrandBrush}" HorizontalAlignment="Left" Margin="10,0,0,0" VerticalAlignment="Top" Grid.Row="2" />
    <RadioButton Name="Filter_Off" Content="Off" HorizontalAlignment="Left" Margin="10,30,0,0" Grid.Row="2" VerticalAlignment="Top" IsChecked="True" />
    <RadioButton Name="Filter_Blur" Content="Blur" HorizontalAlignment="Left" Margin="60,30,0,0" Grid.Row="2" VerticalAlignment="Top" />
    <!--End Radio-->

    <!--ModelNameArea-->
    <Grid x:Name="ModelNameArea" Grid.Row="4">
        <Grid.RowDefinitions>
            <RowDefinition />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Label x:Name="Name_Label" Content="MODEL NAME:" Foreground="{StaticResource SecondaryBrandBrush}" HorizontalAlignment="Left"
```

```
                        VerticalAlignment="Top" Grid.Column="0" Margin="7,0,0,0" />
                    <TextBox x:Name="Model_Name" Text="modelName" HorizontalAlignment="Left" VerticalAlignment="Top" Width="140" Margin="17,0,0,0" Grid.Column="1" />
                </Grid>
                <!--EndModelNameArea-->

                <Button x:Name ="Export_Model" VerticalAlignment="Bottom" Margin="50,0,0,23" Style="{StaticResource␣KinectButton}" Grid.Row="5" Click="Export_Model_Click">
                    <StackPanel Orientation="Horizontal">
                        <Label x:Name="Export_Label" Content="EXPORT␣MODEL" Foreground="White" FontFamily="{StaticResource␣KinectFont}" FontSize="14" Padding="0,0,10,0"/>
                        <Image x:Name="Download" Source="Images/download.png" Width="23" Height="23" HorizontalAlignment="Left" VerticalAlignment="Top" />
                    </StackPanel>
                </Button>
                <!-- <TextBox Name="test_text" HorizontalAlignment="Left" Height="109" Margin="42,124,0,0" Grid.Row="3" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top" Width="209"/> -->
            </Grid>
            <!--MenuArea-->
            <tk:KinectSensorChooserUI Name="sensorChooserUI" HorizontalAlignment="Center" Margin="330,0,330,5"/>
        </Grid>
        <!--LayoutGrid-->

</Window>
```

References

[1] Convolution filter - how to, January 2010. `http://www.roborealm.com/help/Convolution.php`.

[2] How kinect and kinect fusion (kinfu) works, December 2011. `http://razorvision.tumblr.com/post/15039827747/how-kinect-and-kinect-fusion-kinfu-work`.

[3] Helix 3d toolkit - documentation, November 2012. `http://helixtoolkit.codeplex.com/documentation`.

[4] Documentation - alexrj/slic3r wiki, April 2013. `https://github.com/alexrj/Slic3r/wiki/Documentation`.

[5] Print your own yoda at home for $499: 3d printers become affordable, April 2013. `http://www.itworld.com/353914/print-your-own-yoda-home-499-3d-printers-become-affordable`.

[6] Prusa mendel (iteration 2) - reprapwiki, April 2013. `http://reprap.org/wiki/Prusa_Mendel`.

[7] Helmut Grabner Xiaofeng Ren-Kurt Konolige Andrea Fossati, Juergen Gall, editor. *Consumer Depth Cameras for Computer Vision Research Topics and Applications*. Springer-Verlag London, 2013.

[8] Richard A. D'Aveni. 3-d printing will change the world. *Harvard Business Review*, 2013.

[9] Lesliei Gordon. The changing face of 3d printing. *Machine Design*, 2013.

[10] Satoshi Tanaka Soo-Hyun Park Jong-Hyun Kim, Kangsun Lee. *Advanced Methods, Techniques, and Applications in Modeling and Simulation*. Springer Japan, 2012.

[11] Ashlee Vance. 3-d printing spurs a manufacturing revolution. *New York TImes*, 2010.