Bard College
Bard Digital Commons

Senior Projects Spring 2017 Bard Undergraduate Senior Projects

Spring 2017

# Beyond Homographies: Exploration and Analysis of Image Warping for Projection in a Dome

Kai Joseph Malowany
*Bard College*, km3693@bard.edu

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2017

Part of the Graphics and Human Computer Interfaces Commons

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 4.0 License.

## Recommended Citation

Malowany, Kai Joseph, "Beyond Homographies: Exploration and Analysis of Image Warping for Projection in a Dome" (2017). *Senior Projects Spring 2017*. 315.
https://digitalcommons.bard.edu/senproj_s2017/315

This Open Access work is protected by copyright and/or related rights. It has been provided to you by Bard College's Stevenson Library with permission from the rights-holder(s). You are free to use this work in any way that is permitted by the copyright and related rights. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. For more information, please contact digitalcommons@bard.edu.

Bard

# Beyond Homographies:
## Exploration and Analysis of Image Warping for Projection in a Dome

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

By
Kai Joseph Malowany

Annandale-on-Hudson, New York
May 2017

# Abstract

The goal of this project is to provide multiple approaches for warping a flat image to fit the curvature of a geodesic dome, to be presented as an immersive, Augmented Reality (AR) environment. This project looks to develop an algorithmic method of warping any image to fit perspective distortion for a dome-like surface. Despite fairly common usage in planetarium methods and other such shows, there is very little documented method that would allow for the warping of images to fit a curved projection surface. The methods will be explored include using Processing, OpenCV, and fisheye image filters. In addition to the paper, this research will also produce an online library of documents and resources for preforming these warps.

# Contents

Abstract …………………………………………………………………………….. ii

Dedication ………………………………………………………………………….. viii

Acknowledgements …………………………………………………………….... x

1: Introduction …………………………………………………………………… 1
1.1 Background ………………………………………………………………….. 1
1.2 Previous Work ……………………………………………………………… 3
1.3 Motivation …………………………………………………………………… 3
1.4 Review of Literature ……………………………………………………… 4

2: Mathematical Concepts and Construction ……………………………………... 11
2.1 Homogeneous Representation ……………………………………………... 11
2.2 Transformations ……………………………………………………………. 13
2.3 Homographies and Transformation Matrices ……………………………… 15
2.3.1 Matrices and Matrix Multiplication ……………………………… 15
2.3.2 Warping Matrices ………………………………………………… 16
2.3.3 Homographies ……………………………………………………. 17
2.4 Dome Construction ……………………………………………………… 19

Chapter 3: Software and Warping ……………………………………………... 22
3.1 Processing, OpenCV and Syntax ………………………………………... 22
3.1.1 Processing ………………………………………………………... 22
3.1.2 OpenCV ………………………………………………………….. 24
3.2 Warping Strategies ……………………………………………………… 25
3.2.1 Spherical Warping ……………………………………………….. 25
3.2.2 Affine Triangle Warping …………………………………………. 26
3.2.3 Source-to-Destination Warping ………………………………….. 27
3.2.4 Backwards Warping ……………………………………………… 28
3.2.5 OpenCV Warping ………………………………………………….. 28

Chapter 4: Results ........ 30
4.1 Image Selections and Projector Calibration ........ 30
4.1.1 Image Selection ........ 30
4.1.2 Projector Calibration ........ 32
4.2 Preliminary Results and Code ........ 32
4.2.1 Initial Affine Warping ........ 33
4.2.2 Initial Pixel Array Warping ........ 33
4.2.3 OpenCV Implementation ........ 36
4.2.4 Piecewise Affine Warping ........ 38
4.3 Results and Final Projections ........ 40
Chapter 5: Conclusions and Further Work ........ 45
5.1 Final Conclusions and Analysis ........ 45
5.2 Future Work ........ 46

References ........ 50

Appendix A ........ 51
Appendix B ........ 52
Appendix C ........ 57

Project Summary and Information at
Malowanykai13.wixsite.com/seniorproject

# List of Figures

1.1 Paul Bourke …………………………………………..………………………… 9
2.1 Image Transformations ……………………………..………………………… 14
2.3.1 Demonstration of Matrix Multiplication Procedure ……………………………... 16
2.3.2 Homography Warping Transformation Example ………………………………... 18
2.4.1 Construction of the Dome ……………………………………………………... 19
2.4.2 Dome Interior Projector Set-up ………………………………………………... 20
3.1.1 Processing Pixel Array Deconstruction ……………………………………….. 22
3.1.2 Example of Processing Pixel Array Manipulating Color Values ………………... 23
3.2.1 Spherical Filter Warping …………………………………………………….. 26
3.2.2 Affine Triangle Warping …………………………………………………….. 27
3.2.3 Source to Destination Warping ………………………………………………... 27
3.2.4 Backwards Warping ……………………………………………………….. 28
3.2.5 OpenCV Mat Deconstruction ………………………………………………. 29
4.1 Example of Blank Space from curve of Projection Surface ……………………….. 31
4.2 Analysis of Image Choice …………………………………………………….. 32
4.3 Demonstration of Capturing Mouse Position Within Input Image ………………… 35
4.4 Example of Homogrpahy Based 2 Dimensional Warp Using Created Software ….. 37
4.5 Examples of Different Distortion Correction Generated Based on Point Selection .. 38
4.6 Initial Projection of Flat Grid Image onto the Projection Surface …………………. 40
4.7 Piecewise Affine Warping Test ……………………………………………… 41
4.8 Final Rendering Comparison ………………………………………………… 42
4.9 Final Construction and Projections …………………………………………… 43

# Dedication

I dedicate this work to my family, stepfamily, friends, teachers, professors and pets, and to anyone and everyone else who helped become the person I am today.

# Acknowledgements

Thank you, Professor Keith O'Hara, you supported me and coached me through choosing this topic, fleshing it out, finalizing it, and picking up the pieces when nothing went as planned. I would also like to thank the entire computer science department for providing me with the foundation to create this work.

Thank you, Stef, Henry, Charlie, Ani, Phoebe, Cleo and Quinn who supported me and stuck by my side through all 4 years at Bard (and of course Ethan for 3).

Thank you, Race, Bobby, Noah, Quincy, Darren and Ben S. for helping each other and for helping me, and making me feel at home in a new department.

Thank you, Eva-Marie for making senior year by far my best year at Bard and for pushing me to always be my best.

Thank you, Julie, Kristin and Dominique for making me feel at home in work and in school.

Thank you, Ryan Depew for initially putting me on the path to choose computer science.

Lastly, thank you to my Family, for their unconditional love and support and for making me the person I am today.

# 1

# Introduction

## 1.1 Background

The concept of virtual reality first appeared in Stanley G. Weinbaum's short story *Pygmalion's Spectacles* in 1935 [7]. There it was described as a goggle based virtual reality system with holographic recordings of fictional experiences including smell and touch. In the short story Weinbaum writes "*A movie that gives one sight and sound [...] taste, smell, and touch. [...] You are in the story, you speak to the shadows (characters) and they reply, and instead of being on a screen, the story is all about you, and you are in it.*" This may be the first comprehensive model for virtual reality. In the 1950's Morton Hellig wrote of an "Experience Theater" that could involve the user in the experience with all 5 senses, and built a prototype of his design, calling it "Sensorama". In 1978 MIT created a hypermedia [8] virtual reality system called the Aspen Movie Map, which was a crude simulation of Aspen, Colorado in three modes: summer, winter, and polygons. The first two were simply photographs of every possible movement through the town in both

seasons, and the third was a basic graphical representation of the city constructed via early 3D modeling software. Virtual reality continued to appear in popular and mainstream media throughout the 1980's in movies such as *Brainstorm* and *The Lawnmower Man*. In 1991 Sega came out with the *Sega VR* [9] headset for arcade games and the Mega Drive Console, which was the first of what we think of as virtual reality today. VR continued to gain in popularity and accessibly through the video game industry, first appearing in large arcade games and flight simulators, and more recently being moved over to console and PC compatible third party brand VR goggles that can be used at home.

Augmented Reality (AR) arose from VR more recently; AR is an environment that is immersive and interactive but not confined to goggles or a projection device. AR environments are based on the subject's surroundings rather than existing completely in virtual space. The only common example of AR in modern culture is a planetarium, where the show is projected onto a specially designed dome ceiling, which is sometimes with added spheres to act as planets and mobile devices that simulate orbits. AR has existed only very recently in pop culture as well, being featured in the popular Netflix show, *Black Mirror*.

Computer vision is an interdisciplinary field based in computer science. It deals with artificial systems' ability to extract information from digital images and process this information; this lead to the ability to reconstruct images and even corrects for perspective distortion when viewing the image. This project will specifically focus on the area of computer visions the deal with image projection and projective geometry.

## 1.2 Previous Work

We built our research on previous work done in the fields of image projection and image warping. Software such as the one created for this project is used in planetariums to warp images to fit the curvature of the domes onto which they are projected. The goal of this project is to implement two previously used methods of warping images for projection and use them in a new environment. Since there is a lack of work in the field of warping live images fed to the program form a camera, the research for this project has been based on several previous works in the fields of image warping and image projection. Paul Borke [4] has written several papers on image warping for use in projection on the inside of a dome and Codeanticode [10] sites his work in the Planetarium code article. Other sources describe methods of warping images based on matrix multiplication using homogeneous coordinates described in section 2.1.

## 1.3 Motivation

The inspiration for this project came out of melding two ideas. Professor Ben Coonley, in the electronic arts department at Bard College designed and built a geodesic dome intended for use as an immersive virtual reality environment (VRE). Images were to be projected on the inside of the dome, warped so that the perspective gave the illusion of a 3D environment surrounding the subject inside the dome. Initially Professor O'Hara suggested designing an interactive experience or game of some variety for the dome. However I was set on the idea of working with drones after taking a class on robotics the previous semester. As we moved forward and developed our idea more the idea of using the dome stuck. My initial idea was to use a drone to map and render images from its

camera to then be transformed into virtual 3-dimensional maps of the landscape it flew over, however, integration of this idea and the dome proved difficult. We finally decided to push forward with the idea of using the dome for some variety of interactive experience. The shape of the project then began to take more detailed form; we decided to write an image-warping program to manipulate projections so that the perspective inside of the dome was not distorted. There are many different approaches to this task, so several different strategies would be explored. Most of the previous work in the field of computer vision in regards to image distortion and image warping is done with the goal in mind of correcting for distortion in images captured with fisheye lenses or other types of lenses. There exists very little previous work in the field of warping images to a particular distortion factor in order to be projected on a non-flat surface.

## 1.4 Review of Literature

3D projection and augmented reality (AR) software is used commonly today in a variety of fields. The most common use for spherical projection warping is in planetarium light shows [10]. Due to the spherical structure of the planetarium projection screen the images must either be manipulated of projected form multiple different sources to appear undistorted when viewed on the curved surface. In AR projection is less common, usually AR software relies on a device or screen that the manipulated environment is viewed through, such as games like Pokémon Go™ and other mobile apps. One of the only fields in which projection AR is common is electronic and film art – in which abstract images are rendered digitally and projected on the inside of a dome or room to be viewed in an immersive environment. This review of literature will seek to create an overview of the

existing work in the field of image warping and computer projection as well as analyze existing strategies used to warp images for projection purposes. It will also discuss work that has been done with AR systems implementing projective tools and establish a stable foundation of technical knowledge on which we will discuss the approaches and concepts used in this project.

AR is a relatively new concept in the field of computer vision and human/technology interaction, and only very recently has begun to be used for entertainment. In 1997 Ronald T Azuma conducted a survey of the existing augmented reality technology and applications for the MIT press Journal *Presence: Teleoperators & Virtual Environments* [1]. The survey first seeks to define what AR is and how it is to be distinguished from VR and other interactive technologies; the qualifications given are that something that is considered to be AR must:

1. Combine both real and virtual aspects that are related.
2. Be interactive in real time.
3. Be registered and act in 3 dimensions.

Building on this definition the survey looks at why AR is interesting and what we can use AR technology for; AR enhances the users perception, it creates an interactive environment that is not confined to a virtual space and has many, many real world applications. In the medical field AR could be used to augment surgery, to super impose CT or MRI scanned images on the patient to make surgery more efficient, as well as provide real time imagery of surgery to aid in technical training. In the field of manufacturing, schematics and repair manuals could be projected onto the project, giving workers better information on how to fix and build products. In every day life AR could

allow for annotation of the world around us, providing useful quantitative information immediately. Azuma goes on to discuss the applications of AR in entertainment, which is where this project falls. In entertainment Azuma discusses several projects employing AR tactics, including a 1995 performance exhibition using real life actors and digital project environments, and the ALIVE project by MIT where virtual intelligent creatures populate the environment around the user and interact with them.

In the field of computer vision the idea of correcting images based on perspective distortion is not only common, but also very crucial for many applications. J.K. Aggerwal and Shishir Shah discuss the applications of calibration and correction of a fish eye distortion model in regard to computer vision in their 1996 paper *Intrinsic Parameter Calibration Procedure for a (high-distortion) Fish-eye Lens Camera With Distortion Model and Accuracy Estimation* [2]. Their approach was to establish a camera calibration model for removing the distortion from images captured with a fish-eye lens. This procedure is essential for many computer vision applications including robot navigation, stereovision, AR and VR, and robot vision. Using a basic grid pattern they present a calibration method for high distortion cameras based on the effective focal length of the lens, the optical center, one-pixel width on the image plane, and the distortion coefficients present in the transformation matrix. This approach is similar to the one we took in our research in process but the goal is different. Where Aggerwal and Shah seek to correct for the distortion present in an image captured with a distortion heavy lens, we look to implement a perspective distortion warp to images in order for them to be projected in a dome and appear undistorted. They also implement a more physical method of acquiring data on image distortion, using a perforated piece of paper as the

grid in order to more easily obtain the optical center, where most of the methods and image manipulation we conduct are only based on digital parameters.

Next we will look at what algorithms and resources exist in the field of computer vision (CV) in terms of projection and projective transformations (Homographies). Andrea Vadaldi and Brian Fulkerson detail many of these resources in their work on VLFeat –*VLFeat – An Open and Portable Library of Computer Vision Algorithms* [3]. Vadaldi and Fulkerson discuss what resources are available to computer vision researchers and students through VLFeat as well as what format they are in and what specific CV algorithms are available. VLFeat is important to acknowledge for this project as it gives us a definite sense of what resources exist and what aspect of CV have been well documented. VLFeat is similar to OpenCV in its nature as a library of algorithms. OpenCV is one of the main resources we rely on for this project. VLFeat, like openCV, contains numerous easily accessed and easily implementable algorithms for computer vision that have been streamlined into a standard format for simple access (MATLAB for VLFeat, C++/C/Java/python for OpenCV).

One example of a different application of image warping in the field of projection and projective geometry is work done by Paul Bourke. Most of the examples are used to correct for distortion already present in the images so they can be viewed in a natural 100-degree field of view (FoV) on a flat surface. This study done by Paul Bourke in 2004 entitled *Converting a Fisheye Image into a Panoramic, Spherical or Perspective Projection* [4] discusses how to correct for the distortion present in a photo taken with a fisheye lens with the intent to project to photo as a flat panorama. This differs from previously discussed study, Aggerwal and Shah, as Aggerwal and Shah simply designed

a calibration scheme to correct for perspective distortion in images captured with a fisheye camera. Bourke takes fisheye images and creates a 3D grid pattern based on the perspective of the image, thus making it possible to view different parts of the image with the correct perspective. This allows for a different distortion to be present on different parts of the image (such as is needed to unwarp a panoramic photo captured with a fisheye lens). Bourke details that most flat images are viewed with a 100-dgree FoV, this causes complications when dealing with fisheye images that are panoramic in nature and contain more than a 100-degree FoV, if the image grid is warped to fit a flat surface it will still appear to be distorted. Bourke's solution is to treat the camera's position not as a viewpoint but as a point in space and warp each section of the image based on relation to the camera's position in 3D space. See figure from Bourke's paper below.

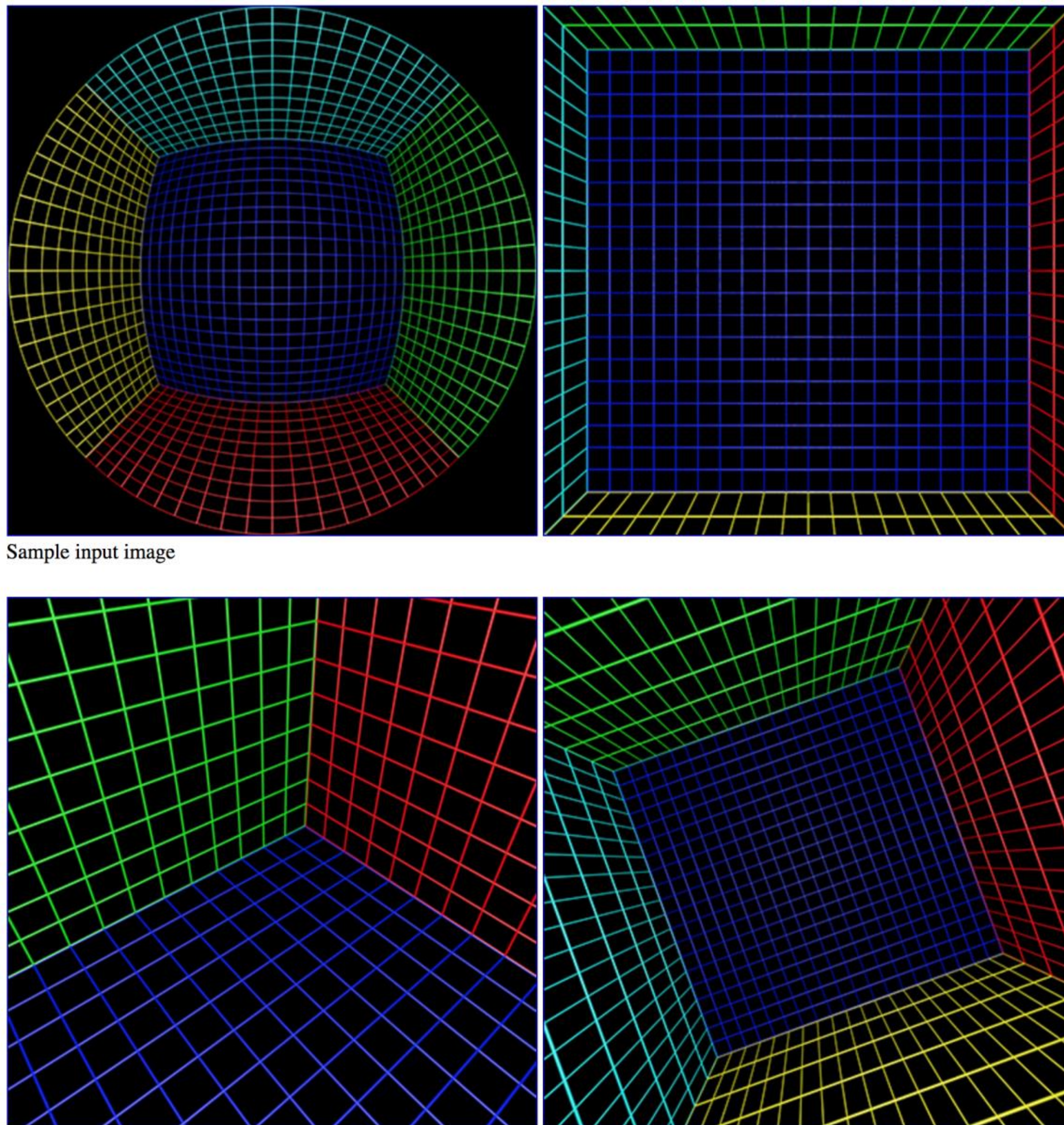



Figure 1. Taken from Paul Bourke's paper [4] showing the different capacities in which the un-warped fisheye image can be viewed

The figure demonstrates the original fisheye image (top left), the un-warped 100-degree FoV image (top right), a view or the bottom right corner adjusted based on the cameras positioning in space (bottom left), and a rotated view with a larger than 100-degree FoV (bottom right). There are several key elements to these transformations, the first is that from the original state the main goal of the distortion correction is to create parallel lines

in the image as can be seen in all but the top left. Once all the lines are straight and parallel other transformations can be performed and the perspective can be changed to any 100-degree FoV segment of the image. The discrepancy between the fisheye image and a regular 100-dgree FoV image on flat surface is important to note as it presents an issue for our goal in the project, namely that any image that looks normal when viewed on a flat surface at 100-degrees FoV will not entirely fit to a spherical warp. This leaves us two options, either warp what we have of a normal image and correct for any discrepancies that arise (Figure 4.1), or use images that are panoramic in nature so that there is a natural wrap around the viewer (Section 4.1).

# 2

# Mathematical Concepts and Construction

This section will discuss some of the basic mathematical concepts that make up this project, as well as some more advanced projective geometry concepts and models that will be used in image warping. It will also describe the basic dome construction and the projector set up that was used throughout the project.

## 2.1 Homogeneous Representation

Homogeneous coordinates were fist introduced by August Ferdinand Möbius, and would become a powerful tool in the field of computer vision and projective geometry. The homogeneous approach is an alternative to the standard Euclidian, or *inhomogeneous* approach that is faced with some limitations. One of the limitations of the Euclidian system is that it allows for only linear transformations that fix the origin. Another limitation is that it does not have a finite representation for a point or line at infinity. The homogeneous method solves this issue by using an extra dimension. This extra dimension allows for representation of points and lines at infinity. The homogeneous method also allows for the representation of projective transformations as matrices, which is how we

will refer to them in the rest of this paper. This offers an easy and convenient way to preform such transformations.

**Definition 2.1.1** Let P be a finite point, and (x, y) be its representation in Cartesian coordinates. A homogeneous representation of point P is any point (x, y, w) $w \neq 0$. The last coordinate w is called the homogenous coordinate. A homogeneous representation of a finite point has a non-zero homogeneous coordinate, and a point at infinity has a homogeneous coordinate of zero.

**Example 2.1.2.** Let P1 be a two dimensional point with Cartesian coordinates (x, y). A possible homogeneous representation of P1 is (x, y, w), or, more generally, (wx, wy, w) for any $w \neq 0$. Let P2 be another point in two dimensions with homogeneous representation (x, y, w). P2 can be represented in Cartesian coordinates as $(\frac{x}{w}. \frac{y}{w})$.

**Definition 2.1.3.** Let l be a straight line in two dimensions defined by the equation $ax + by + c = 0$. Then (a, b, c) is the homogeneous representation of line l.

Again the homogeneity in this representation comes from the fact that (va, vb, vc) represents the line vax + vby + vc = 0, which is the same as line ax + by + c = 0 represented by (a, b, c), for any $v \neq 0$.

**Definition 2.1.4.** The degrees of freedom of a system are the number of independent parameters that define the system and are free to vary.

**Example 2.1.5.** For any Cartesian point in two dimensions the parameters are the x and the y coordinates. Thus any 2D point has exactly 2 degrees of freedom. Now, recall that any line in defined by some equation of the form $ax + by = c$ ($ax + by + c = 0$). Where a and b are coefficients to the variables x and y and c is a constant coefficient. However, even though the line's equation has 3 variables there are only 2 unique parameters that the line varies by, which are the slope and the Y intercepts commonly represented as m and b in the equation $y = mx + b$.

## 2.2 Transformations

This section of the paper deals with graphical transformations on 2D images. In the field of computer vision, transformations are functions of pixels, represented by x y coordinates that return new coordinates for each given pixel of input. Some simple types for transformations that can be preformed on pixels include translation, rotation, affine, and scaling (See figure 2.2.1) These transformation functions form a basis for more complicated transformations that can be preformed as a set of these simple functions, including projective transformations (homography). The process of applying any of the transformative functions to some image is called warping.

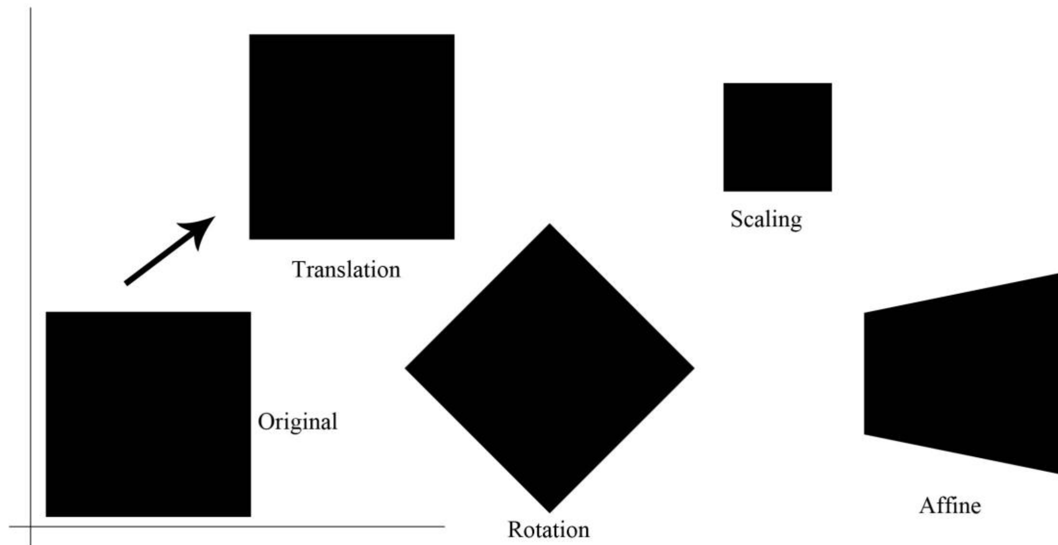



Figure 2. Shows a physical representation of each of the listed transformations on a Cartesian coordinate plane

Next we will discuss some of the fundamental transformations of two-dimensional images listed above and in figure 2.2.1. For each transformation listed we provide a definition and a matrix notation that offers a convenient way to apply them to any given 2D image. Since the transformations discussed in this section do not preserve the origin point (the pixel at (0 ,0)), matrix notation of them is only possible when we use homogeneous coordinates as discussed in the previous section.

**Definition 2.2.1.1** Translation is a transformation that shifts any given input image by a vector to another point in the 2 dimensional Cartesian plane.

**Definition 2.2.1.2** Rotation is a transformation that rotates any given input image around the centermost point of the image.

**Definition 2.2.1.3** Scaling is a transformation that affects the size of any given input image by a constant factor for all the image pixels.

**Definition 2.2.1.4** Affine is any transformation on any given input image that preserves parallel lines and distance ratios between points on a straight line in the source image. Translation, rotation, and scaling are all examples of affine warps.

## 2.3 Homographies and Transformation Matrices

### 2.3.1 Matrices and Matrix Multiplication

**Definition 2.3.1** A matrix is a rectangular array of number, variable, symbols, expressions or equations arranged in rows and columns. The dimensions of a matrix are denoted with the number of rows first then the number of columns like (rows x columns). Vectors are (3x1) matrices. The elements of the matrix define operations that can be preformed.

**Definition 2.3.2** Matrix multiplication, or matrix product, is a binary operation that produces one matrix form two by multiplying together the components of each. Let M be and n x m matrix that is being multiplied together with matrix N, which must be m x p, where p is arbitrary. The product of M and N, L will be a n x p matrix. Note that it is only possible to multiply N and M together if they share the value m, which must be the number of columns for N and the number of rows for M. see figure 2.3.1 for a more detailed example of how to obtain each value in the product matrix.

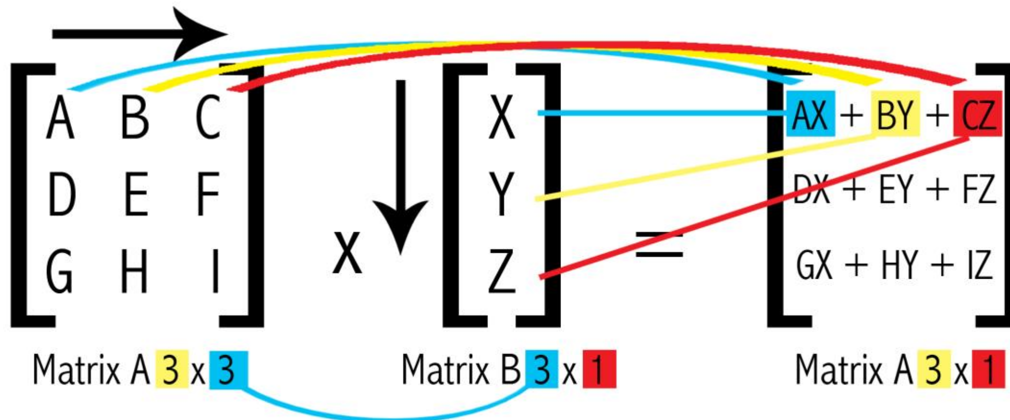



Figure 2.3.1 Shows the process of multiplying 2 matrices together. Note for the multiplication to be possible the first matrice's column number must be equal to the second matrice's rows value and the product matrix inherit the first's row value and the second's column value. For each row in the product matrix the values of each entry in the first row of matrix A are crossed with the values of the first column in matrix B and then the sum is taken.

## 2.3.2 Warping Matrices

Matrices and matrix multiplication can also be used to represent transformations of images in virtual space for projection. Typically transformations are represented as matrices for ease of multiplying the x and y coordinates of each pixel in an image by the contents of the matrix. The following matrices matrix can be used to represent each of the above transformations form section 2.2 assuming that the images coordinates are in homogeneous representation:

Translation can be represented by a 2x3 or 3x3 matrix taking the form

$$\mathrm{T} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \end{bmatrix}$$

$$\mathrm{T} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

For some change in the x position Δx, and some change in y position Δy.

The result of the matrix multiplication on a Cartesian coordinate point is

$$\begin{pmatrix} x + \Delta x \\ y + \Delta y \end{pmatrix} \text{ And } \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{pmatrix}$$

Rotation is typically represented by a 2x2 matrix but can also be shifted to a 3x3 matrix if the points are in homogeneous representation.

$$\mathrm{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathrm{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scaling transformations take the form of a single integer scaling-factor and are not effected by what type of coordinate system is used.

These transformations can be combined to affect the image in other ways too, usually done by combing the transformation matrices. An example is the following matrix that represents a scaling and rotational transformation.

$$A = \begin{bmatrix} \mathrm{s}\cos\theta & -\sin\theta & 0 \\ \sin\theta & \mathrm{s}\cos\theta & 0 \\ 0 & 0 & s\,1 \end{bmatrix}$$

## 2.3.3 Homographies

**Definition 2.3.3** A homography, also called a projective transformation or perspective transformation, is a transformation of a two-dimensional source image I to another two-dimensional image I' in the same coordinate plane, such that all straight lines and distance ratios between pixels in I are preserved in I', it is represented by a 3x3 matrix

$$\begin{bmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ m_7 & m_8 & m_9 \end{bmatrix} \times \begin{bmatrix} X_n \\ Y_n \\ 1 \end{bmatrix} = \begin{bmatrix} X_d \\ Y_d \\ w \end{bmatrix}$$

Figure 2.3.2 an example of a matrix tranformation applied to the homogeneous coordinates Xn and Yn. The result is the distorted coordiantes for that specific point. when applied to all pixels of an image the 3 x 3 matrix will prodcue a distorted (or transfromed) image.

Homographies are use to map an image on a plane, in this case a flat plane as you would normally view on laptop or any other flat surface, to another image with a perspective distortion, such as viewing a billboard image from an angle.

There are two distinct types of homography based warping: forward and backward warping. Forward warping takes each pixel in the source image and maps it via the homography matrix to a location in the transformed image. This presents potential difficulties with large distortions because there can be gaps in the image since there is nothing to prevent multiple pixels in the source image from being mapped to the same location in the destination image. Backwards warping differs in that instead of iterating through the pixels in the source image it iterates through the pixels in the destination image and fills each one with color based on the corresponding pixel in the source-image. This ensures that all of the pixels in the destination image are filled and no two source-image pixels are mapped the same destination-image pixel, however it can also run into difficulties when the distortion is large because pieces and detail form the source-image can be lost.

# 2.4 Dome Construction

This section will detail the construction of the dome on which the final, warped images are to be projected.

The dome is constructed of 40 cardboard triangles clamped together and resting a top a custom build wooden base. 15 of the triangles are 35”x35”x35” and 25 of them are 31”x31”x35” as detailed in figure 2.3.1.

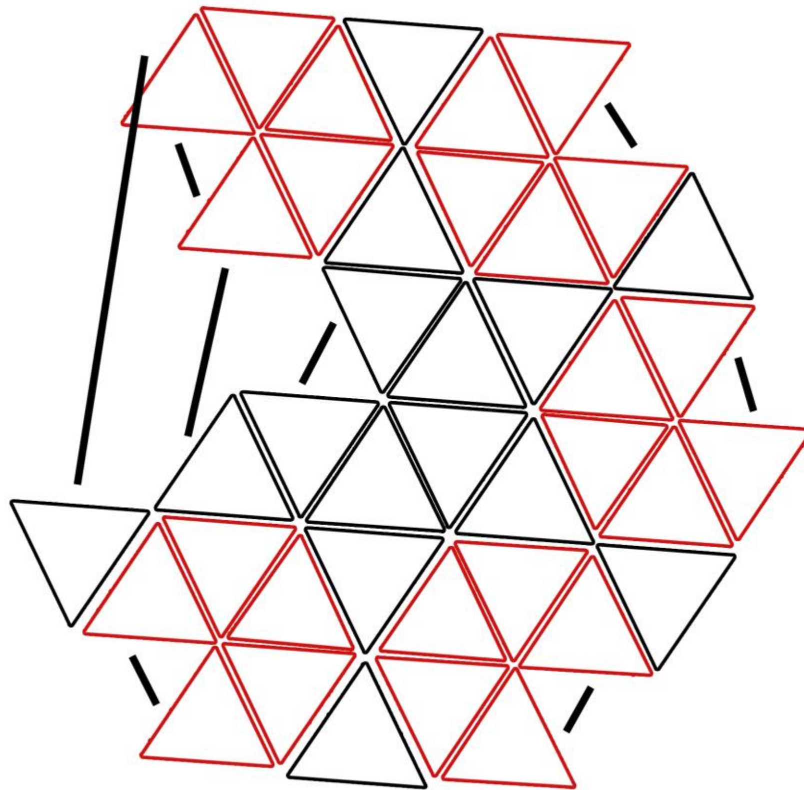


**Figure 2.4.1.** details the cinstruction of the dome onto which we project our images. the triangles depicted in black are 35”x35”x35” and the triangles depicted in red are 31”x31”35”. Each black line connecting two of the triangles indicates where a vertex would be connected if the dome were to not be depicted in 2 dimensions.

Inside the dome we set up a spherical mirror to bounce the projected image off of and to save space within the dome another mirror that reflects the projected image up to the spherical mirror, which in turn bounced the warped image up on to the inside of the dome surface, as shown in figure 2.4.2.

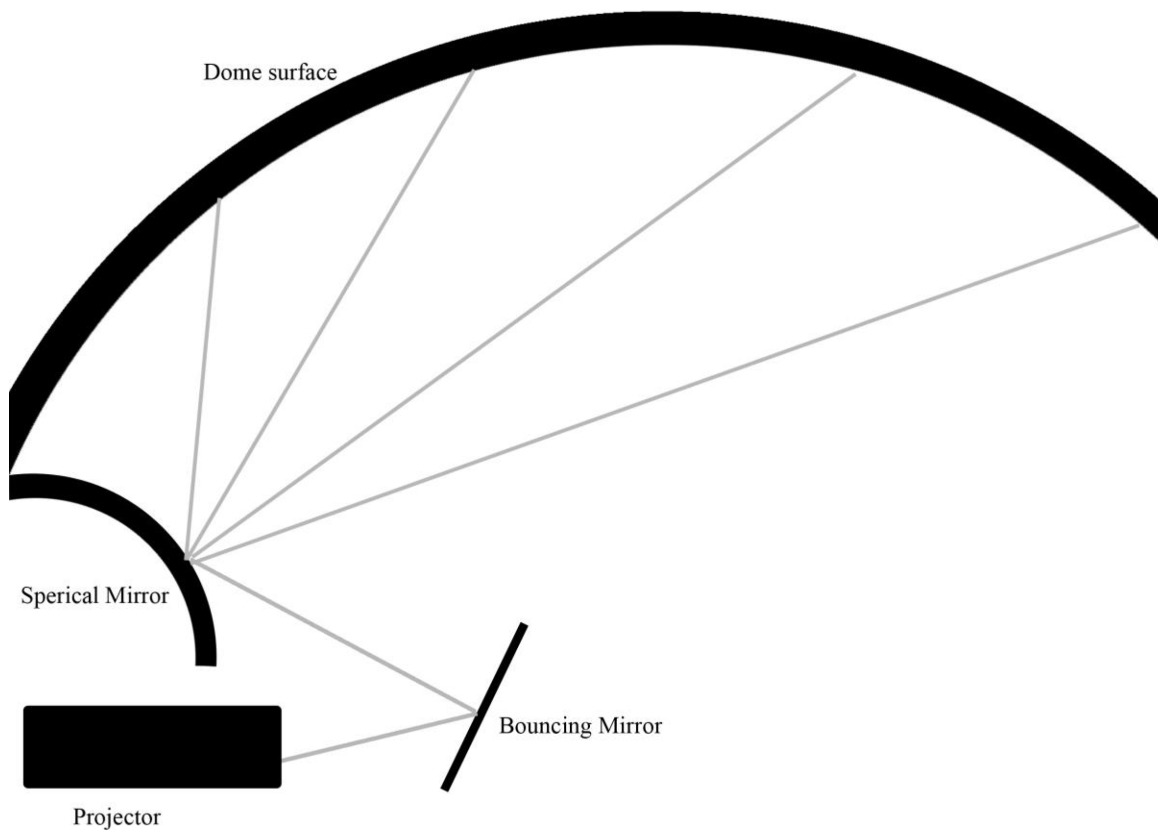
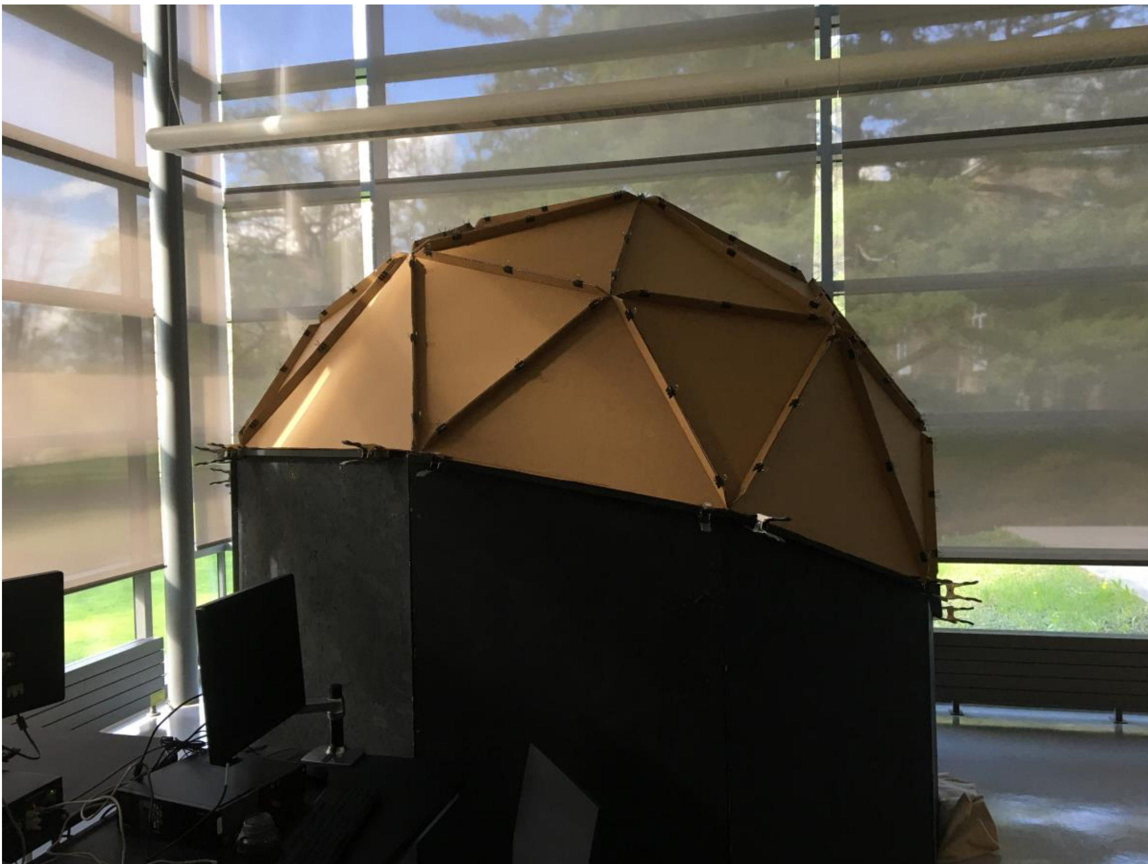



**Figure 2.4.2.** Depicts the view inside the dome and the setup used to save space as well as effectively bounc the projectred images onto the inside of the dome. the projector sits on a small table underneath the spherical mirror and the projected images bounce first off of a small flat mirror angeld so as to minimize distrotion.

This set up, while convenient does add a second layer of distortion to the image. Now we have to compensate for the distortion of projecting a flat image onto a curved surface, as well as a surface that is angled away from the projecting lens. As you can see in figure

2.4.2 the dome surface angles away from the projector causing the top of any un-warped image projected on the surface to be stretched out. Therefore our correction warp needs to be two-fold both correcting for the domes curvature and the angel at which the image hits the surface of the dome.

The dome was designed and built by professor Ben Coonley of Bard College's electronic arts department for an exhibit at the Whitney Museum of American Art title *Dreamlands: Cinema and Art 1905-2016* [15]. Refer to the website listen in the table of contents for more information.



# 3

# Software and Warping

## 3.1 Processing, Open CV

### 3.1.1 Processing

In order to correctly implement the projection and graphics for this project we decided to use and IDE called processing coding in Java. The main interface utilized for the code aspect of the project is Processing Pixels. Processing Pixels is a package built into the Processing IDE; it allows the user to manipulate the pixel of any image via a pixel array (see figure 3.1.1).

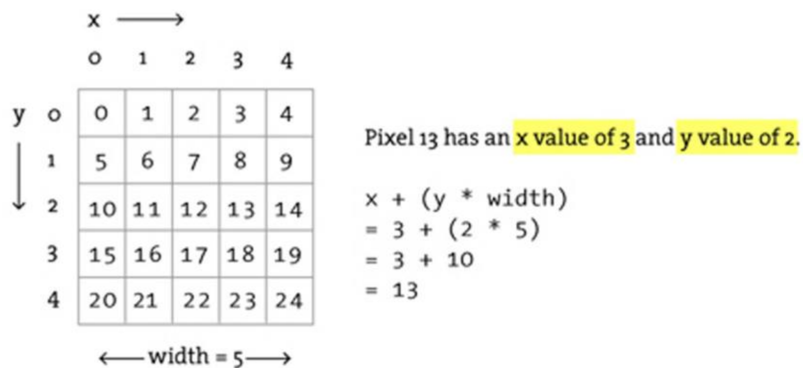



Figure (3.1.1) depicts the way the processing pixels package interacts with the pixels of an image via one-dimensional array. Each pixel's position in the arrya is detemined by adding the x coordinate to the y coordiate time the width of the image in pixels.

The pixels array is a one-dimensional array, where each pixel's position in the array is denoted by the x value plus the y value multiplied by the width of the image. This sets it up so that each row of pixels is in order one after the other with the right most pixel of the previous row being adjacent the left most pixel of the current row. As you can see in figure 3.1.1 pixel 5 (position 4) is directly adjacent to pixel 6 (position 5) in the array. However in the actual source image pixel 5 and 6 are not adjacent. In order to manipulate pixels in the destination image pixels array, the pixels within the array must have their RGB (Red, Green, Blue) values shifted based on pixels from the source image. The destination image's pixel array is then updated and the destination image is drawn based on the new pixel array. Figure 3.1.2 shows a very basic example of pixel manipulation using the processing pixels array.

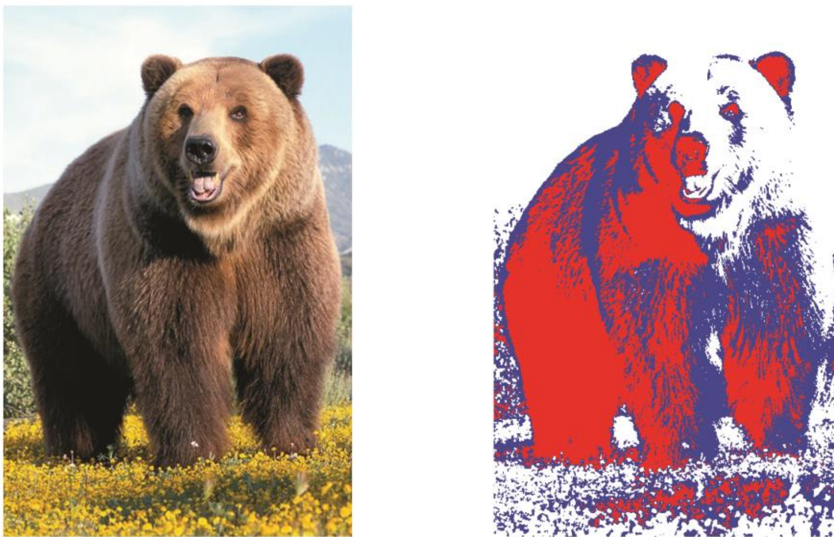


Figure 3.1.2. The result of the following code - using the pixel array to assigning colors based on brightness values.

```
PImage source;   //source image
PImage destination; //destination image

 void setup(){
   source = loadImage("Bear.jpg"); //load an image from a file
  size(source.width, source.height); //size of canvas
  destination = createImage(source.width, source.height, RGB);  //create a blank image as the destination
}

void draw(){
  float redThresh = 85; //establish the parameters for our image filter
  float blueThresh = 170;

  source.loadPixels(); //we want to look at both images pixels
  destination.loadPixels();

  for (int x = 0; x < source.width; x++){ //loop through pixels and create the 1D pixel array
    for (int y = 0; y < source.height; y++){
      int loc = x + y*source.width;

      if (brightness(source.pixels[loc]) <= redThresh){ //test the brightness and assign approproate color
        destination.pixels[loc] = color(2,23,129); //red
      } else if (brightness(source.pixels[loc]) > redThresh && brightness(source.pixels[loc]) <= blueThresh){
        destination.pixels[loc] = color(191,10,10); //blue
      } else {
        destination.pixels[loc] = color(255, 249, 195); //white
      }
    }
  }

  destination.updatePixels();  //we cahnged the pixels in the 'destination' image

  image(destination,0,0); //display destination image
}
```

### 3.1.2 OpenCV

OpenCV is a resource library used for computer vision and image manipulations. OpenCV stands for Open Source Computer Vision. Using the OpenCV library allows us to implement data types and other resources that are not otherwise available in Processing. Although primarily designed and written in C++ there are versions that work with Java (what we are using), Python and MATLAB.

## 3.2 Warping Strategies

There are many different ways to go about warping an image for projection. In this section we will discuss the methods of warping that were attempted and why some failed. For more detailed analysis of some methods see section 4.2.

### 3.2.1 Spherical Warping

One method often employed in planetarium style projection is something we will refer to as spherical warping. Spherical warping means taking any image as input and warping it uniformly to the surface of a sphere. To do this we used a processing graphic module called "planetarium" which rendered an environment four times from four different perspectives, one form each side of the sphere in order to create the illusion of a 3D sphere in the image. The idea was to use the same math and multi-rendering process to warp an input image to fit the curvature of a sphere. Two problems were encountered with this strategy, firstly that, as discussed before, the dome is not a perfect sphere and the projection surface is angled in relation to the projector, this angle creates a vertical stretch distortion that a basic spherical warp would not correct for. Second was the fact that the domes surface is not oriented exactly inline with the camera. So the apex of the dome is not the center of the projected image, which presented an issue for the spherical warp because the environment generated by the multi-rendering approach required the apex of the curvature to be in the center of the image.

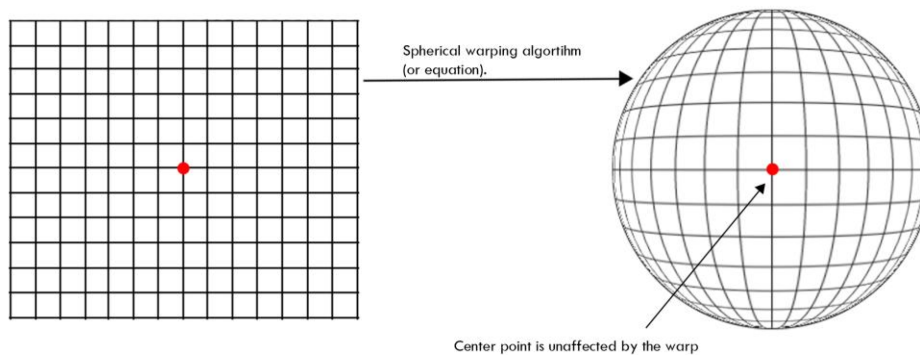



Figure 3.2.1 show the basic way of rendering an image based on a spherical distorntion. usually this code will render and image multiple times to account for multiple viewpoints in a spherical projection and accoutn for the overlap - somtimes including image stitching methods to piece togehter the differenct perspectives.

### 3.2.2 Affine Triangle Warping

As discussed in section 2.2.1 an affine transformation is any transformation of an image that preserves parallel lines and distance ratios between pixels in straight rows or columns in the source image. In order to implement this method successfully first we would need to break the source image into triangles that would then be warped to fit the perspective distortion of each section of the dome as depicted in figure 2.4.1. A grid pattern would be projected onto the projection surface of the dome and the end points of each of the triangles would be identified. Next the source image would be marked with the end points of the triangles and each would be distorted via affine transformation to fit the perspective distortion of the dome face. However the main problem this method presented us was how to separate the source image in the appropriate triangles corresponding to each triangle on the face of the dome. Since the end points on the projected image would also be distorted there was no direct way to find the un-warped end point locations on the source image.  While simple in the practice of only using affine warps, it is also inefficient, as multiple warps have to be preformed for each image render.

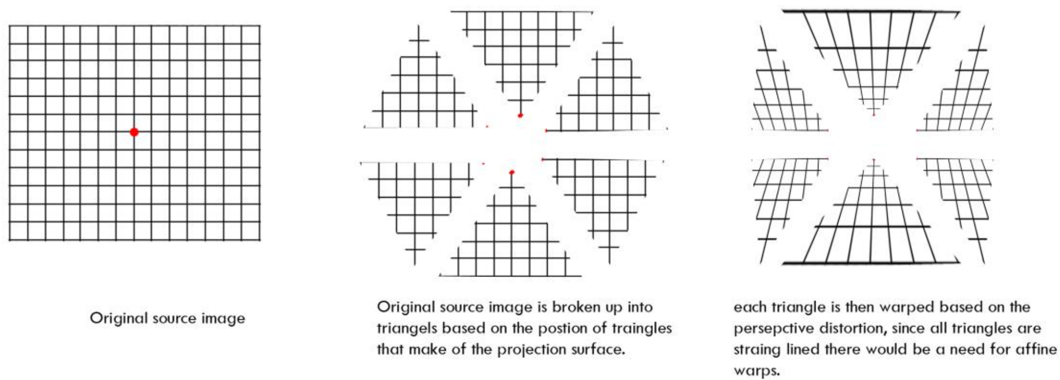



Figure 3.2.2 shows a warping method that is unique to this project. due to the construciton of the dome we are using we have the option of dviding any source image up inot the corresponding triangles to where weach piece of the domes lies when the image is projected. then each triangle is warped based on the perspective distortion for that triangle (and affine warp) and the image is peiced back together.

## 3.2.3 Forward Warping

A program would be written that would iterate through each pixel in the source image and map it to a new location in the destination image. The main issue with a forward approach is that there is no guarantee that the same pixel in the destination image will not be filled by multiple from the source image. This also presents the potential of gaps appearing in the destination image and leads us to the conclusion that the destination-to-source warping strategy (3.2.4) is more advantageous.

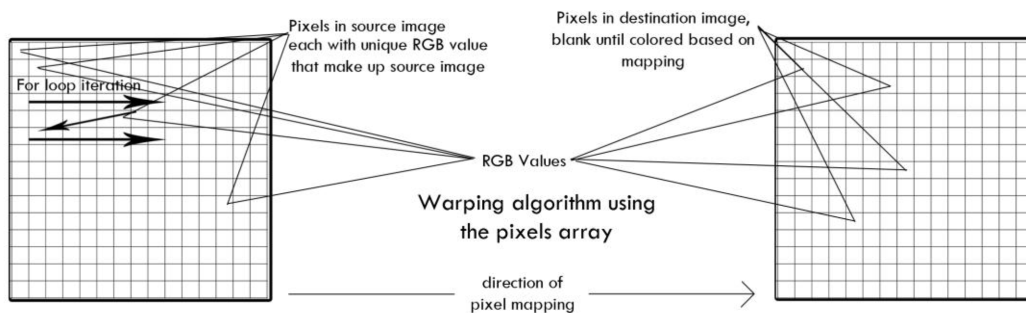



Figure 3.2.3 shows the method of “forward warping” where each pixel in the source image is mapped to a pixel in the destination image. when a pixel is “mapped” it means that the RGB values of each pixel in the source image are assigned to some pixel in the destination image based on the warping equation. this can be simple like flipping and image upside down or it could be a quadratic equation resulting in a fisheye like distortion.

## 3.2.4 Backwards Warping

Backwards warping is the intuitive opposite of forwards warping. A loop iterates through each pixel in an existing but empty destination image and fills each with RGB values corresponding to a pixel in the source image. This strategy eliminated the potential of gaps or pixels overlap in the destination image as every pixel is covered by the loop as needed. The only significant draw back of the strategy is that sometimes pixels form the source image can be ignored is they do not map to any pixel in the destination image, we have found that this is almost always a non-significant effect to the image.

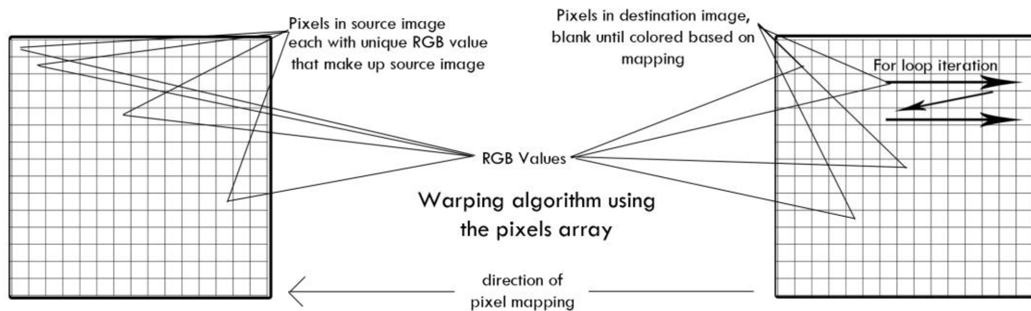



Figure 3.2.4 shows the method of "backward warping" where each pixel in the destination image is iterated through and assigned a color value from a pixel in the source image based on the mapping equation.

Both of these pixel array based warping strategies also run into the same trouble as the spherical filter warping does, since any algorithm or equation would have to be based on the x and y coordinates of each pixel. The warping factor would thus be based on the distance form the center of the image for each pixel, so it would only be able to produce spherical images as well.

## 3.2.5 OpenCV warping

The last method of image warping we will discuss uses the OpenCV library available online. The OpenCV library in processing has built in objects called Mat

(matrix) that allows us to simply give a method (*getPerspectiveTransformation*) an input of PVectors in 2 arrays and it will output the matrix used to acquire the specific perspective distortion between the two arrays of points. This matrix is then used in the warpPerspetive method to apply that specific distortion to an image also given as input. OpenCV allows for a much easier interface with the warping methods, however is does present the challenge of acquiring a matrix-based transformation that is capable of producing a spherical warp.

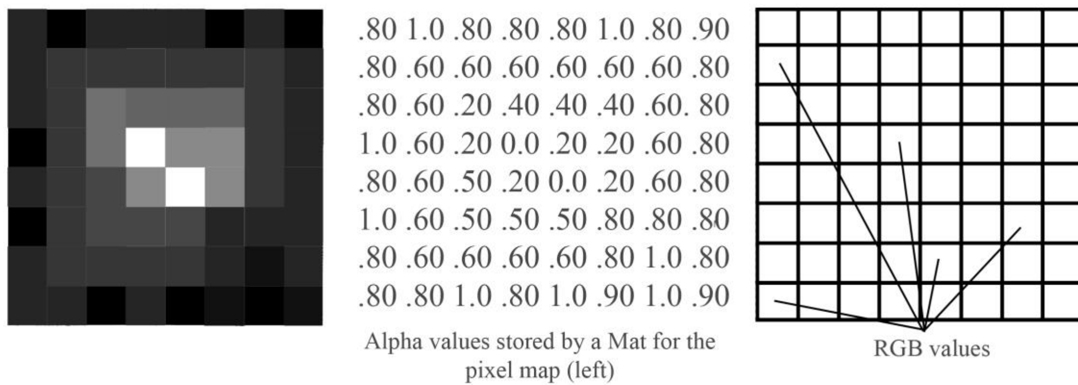



Figure 3.2.5. (left) A a basic sketch of the pixels in and 8x8 black and white pixel grid. (center) alpha brightness values (0-100) stored in an OpenCV Mat object. (right) Sketch of the RGB values stored for each pixel in a Mat object.

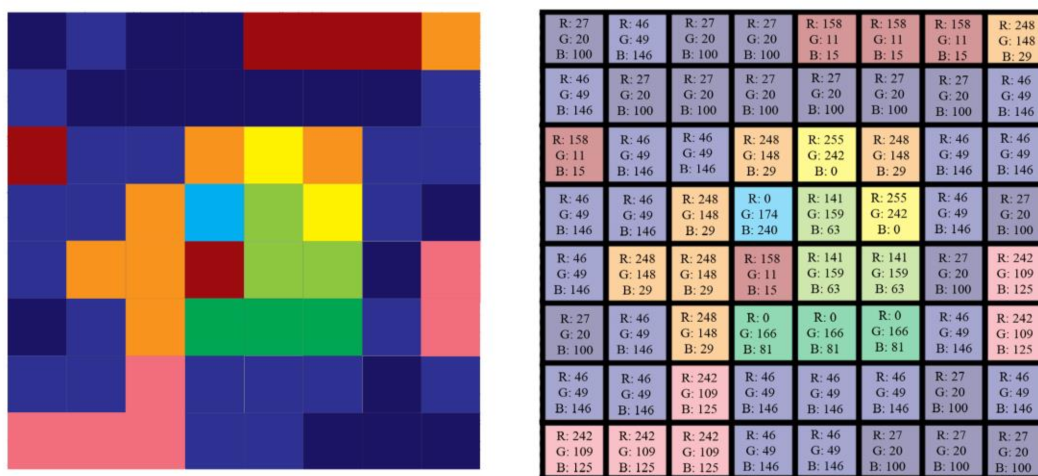



Figure 3.2.5.1. (left) A a basic sketch of the pixels in and 8x8 color grid smaple. (right) A more detailed view of the RGB values stored in a Mat object in OpenCV. each values ranges from 0-255.

# 4

# Results

## 4.1 Image selection and Projector calibration

Before moving to the results of the image warping and projection there are several issues and factors that need to be discussed.

One of the major issues that we encountered while setting up the projection was choosing image that when warped would fit the shape of the dome projection surface. Due to the limitation of the corrections that can be made on image in order for them to not appear stretched or deformed even with correct perspective distortion image shape and field of view (FoV) plays a significant role in how successful and projection is.

### 4.1.1 Image Selection

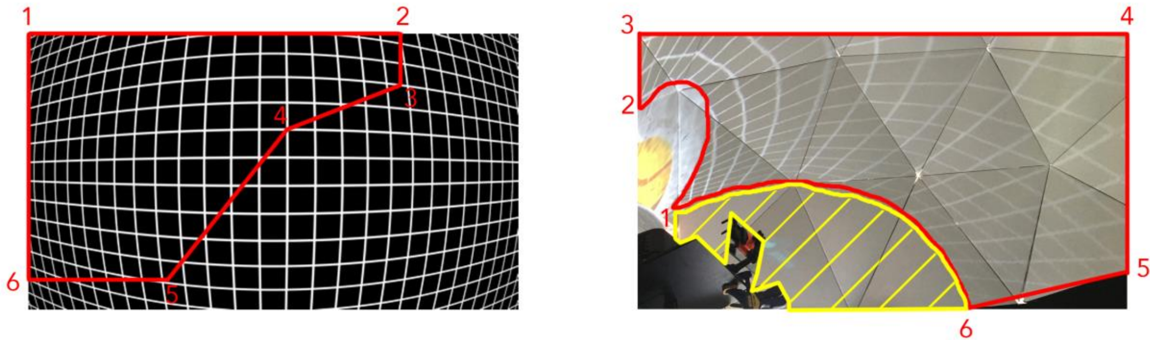



**Figure 4.1** shows an example of the discrepencies between the face of the dome and the flat surface of the viewing plane of a 100-degree FoV image. The shaded yellow region of the right image shows where the curvature of the dome causes an issue in porjection. The points are numbered in both the right and the left image showing where each is in the warped and unwarped version. the width of the the image on the left directly correlates to how much blank space there is on either sid of the distroted image when projected on the dome.

As shown in figure 4.1 a regular image (1,200px X 675px as displayed on a computer screen) used as an example of a pre-rendered warped image, when projected on the dome surface with no other distortion shows large blank spaces on either side of the image. These blank spaces are indicative of discrepancies in the field of view of the subject in the dome, an image that would be better suited to correct for something like this would be any image with a wider aspect ratio. However, if we use an image with wider aspect ratio that is still intended for a flat surface we run into the issue of having an immersive perspective in the dome when the original image was not immersive. The obvious solution to this problem is to use panoramic images that are taken at a wider-than 100-degree FoV. Using a panoramic image would allow for a more immersive experience in the dome as well since a panoramic image is taken from a single point in space revolving around the point in 100+ degrees. This effectively creates a circular plane around the source point of the image that can be substituted with the dome surface, which is also curved and then the other distortion present can be corrected via software.

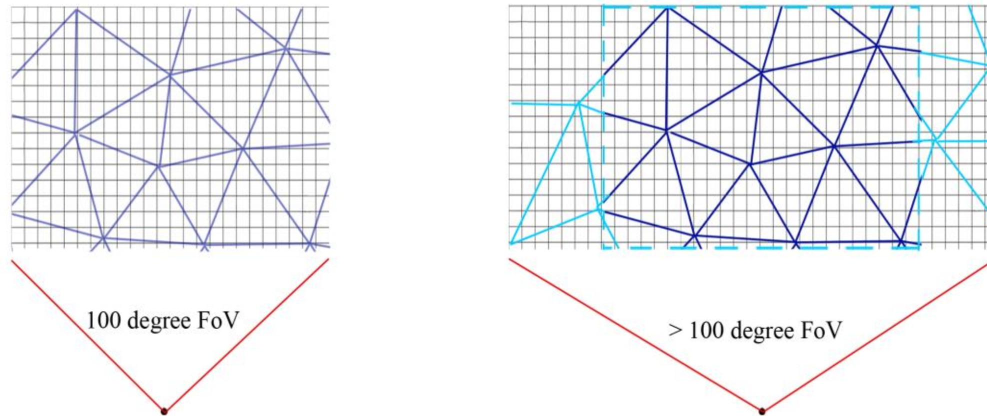



Figure 4.2. Shows the differnec in projection surface coverage between a regular 100 degree field of view (FoV) image (left) and a greater than 100 dgree FoV, or panoramic, image. Note that the panoramic image has a much wider coverage area and thus a more immersive view when in the dome. The extra coverage of the panoramic image is highlighted in light blue.

While simply using panoramic images sounds good in theory there are some spatial limitations based on the physical set up of the dome and the projection surface's distance from the projector. The camera being used in this project for capturing images projected on the dome surface is only equipped with a normal lens, in order to capture a full panoramic image, we would need a wide angle or greater than 100-degree FoV camera. Another spatial issue is the projector's distance from the projection surface – due to the close quarters inside of the dome only a portion of the computer screen can be viewed on the projection surface. These factors lead to choosing a 900x600 pixel aspect ratio for projection, so the entire image can be manipulated and viewed.

### 4.1.2 Projector Calibration

Due to the set up of the projector in the dome and the image being bounced of off multiple mirrors there is a much larger distortion factor from minor shifts in the projector placement. Before choosing the images to calibrate the warping matrix it was essential to correct for any variation in projector placement, this includes lateral shifts and distance

from the flat mirror. The issue of switching projectors midway through the project forced us to deal in depth with projector positioning and the effect it can have on image distortion. Due to technical difficulties that lead to the switching of projectors there was also a 3-5 week period in which access to projecting images in the dome was not possible, forcing the use of virtual simulations and only proof of concept test for the software.

## 4.2 Initial projections and Code.

As described previously there are several different forms that the image-warping piece of this experiment can take. While all revolve around images being projected on a non-planar surface, namely the inside face of a dome, they each take different methods of distorting the image to fit the projection surface. As described in section 3.2 the five basic categories of warping that were considered are spherical warping, affine triangle warping, forward pixel warping, backward pixel warping, and OpenCV based warping. In the initial stages of set we were able to eliminate spherical warping, as we knew the dome was not a perfect sphere and thus there would be far too much disparity between the projected image and the projection surface.

### 4.2.1 Initial Affine Warping

Initially affine warping was also eliminated because there was no clear way to separate the flat images into triangles corresponding to the dome's structure. Due to the distorted nature of our projection set up the image would have to be marked with the end points of each triangle while being projected on the dome surface. While this part is

entirely possible the next stage presents a problem that would initially double the workload. First the image would have to be marked at the corresponding points on a flat surface, or we would have to implement some method of unwarping the image captured on the domes surface, once this image was flat we would be able to divide it into the corresponding triangles and run each triangle through a perspective distortion to then be recompiled and projected once again. However we still had to address not having a method of unwarping the image, as all of the previous work done on distortion correction in projected images relies on symmetry and the distortion coming from the lens with which the image was captured.

### 4.2.2 Pixel Array Based Warping

Next we looked at the implementation of pixel array based warping. This would include both forward and backward warping using the pixel array in the Processing IDE. The base concept would be to create a method that calculates the distortion between a flat grid image and the same grid image projected onto the dome. This distortion factor would be calculated as a matrix.

We wrote a method taking 2 PVector arrays, each of length four, called *getTransform()* (adhering to basic Java naming conventions). Each PVector array contains 4 points from the flat image and the distorted image respectively. Then a distortion factor is calculated in the form of a matrix. First each of the input arrays is converted into a corresponding matrix representing the proportions of each image, ideally the points chosen would be as close to each of the corners of each image as possible in order to get the most accurate picture of the frame of each image. Next the eigenvalues

are calculated and the final transposition matrix is created from the smallest eigenvectors. This method produces a 3x3 homography initially but could, in theory, be scaled up to create larger, more complicated matrices for non-homographic transformations. Next we turned our attention to how to get the PVector arrays from each image and how to render the transformed image form the transposition matrix. First we addressed the method used to get the PVector points from each image, this is later used in the OpenCV warping technique as well. Using the built-in *void mousePressed()* function in Java a method was written to capture the mouse position as a PVector of x and y position wherever the mouse was clicked.

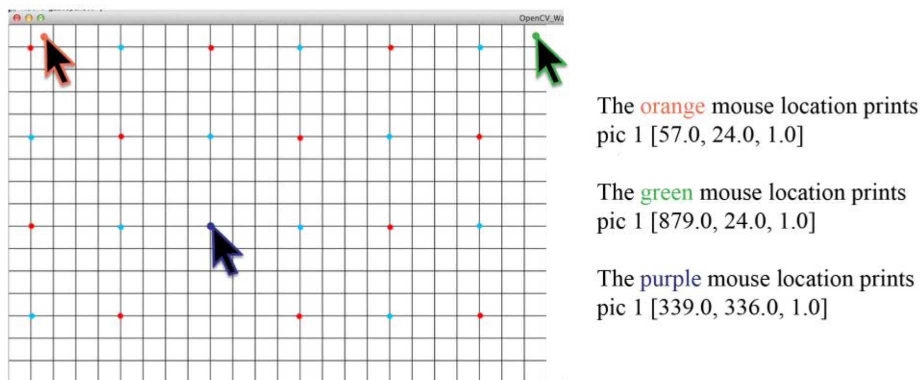



**Figure 4.3** shows half of the rendered iamge used for sourcing the points that will be input into the PVector array. Whenever the mous is clicked if it is within the image on the left if with record the corrdinates int the source array and print them out with the stamp pic 1, if it is within the image to the right (not shown) it will record the coordinate into the destination array and print them out with the stamp pic 2, all points used are in homogeneous representation.

This PVector is then automatically sorted into the appropriate array based on which image it is in. Once each array size is equal to four the program call the *getTransform()* method using the newly filled PVector arrays as input. Next, using the processing pixels array defined in section 3.1.1 each pixel in the form of a homogeneous coordinate PVector is fed though the matrix and its new x and x position is dictated based on the matrix calculated by the *getTransform()* method. The code that was used to construct this

software was based on a lab assignment form Professor O'Hara's computer vision class [Appendix C]. The problems we ran into with this method are centered around the complexity of a non-affine warp being implemented on the image. As defined before an affine wrap is any transformation of a grid that preserves straight lines in the image. Since the Dome's surface is curved the lines needed in the source image are distinctly not straight and this presents a challenge when using a traditional matrix for warping the image.

### 4.2.3 OpenCV Implementation

The shortcomings of the previous strategy lead us to the computer vision library online, OpenCV. OpenCV has a structure called a Mat, which functions similarly to a traditional matrix, but has an entry for every single grid value in an OpenCV image object. Mat represents a n-dimensional dense numerical array, used to store values, matrices and equations. The complexity of a Mat object in OpenCV allowed us to achieve a much more accurate warp, having a data entry for each of the pixels from the source image, dictated by passing the Mat generating function the size of the source image as an argument, was a great advantage. This strategy worked very similarly to the forward pixel based warping. Initially the images are declared and converted into OpenCV objects so they will be able to manipulate them using the algorithms later. Next we use the same method to acquire PVector coordinates from both a flat image and a warped image as described above. The same PVector arrays are input into a method called *getTransform()* as well. The PVector arrays are then converted into processing

based *Point* objects, which are then converted into MatOfPoint2f objects one each for the warped image and the source image. The MatOfPoint2f object is a type of Mat with the image processing (imgproc) library of OpenCV, then a method called *imgproc.getPerspectiveTransform()* taking each of the two MatOfPoint2f objects as arguments, calculates the differences between the two Mat objects and outputs that difference as a third Mat object. This is done by calculating a 3x3 matrix based on four input points in each image and then formatting a Mat object based on the 3x3 matrix. Once the Mat object is created it is simply run through a method the converts the Mat and an input PImage to a new PImage with the perspective distortion represented by the Mat object. This PImage can then be drawn by Processing's *Draw()* function.

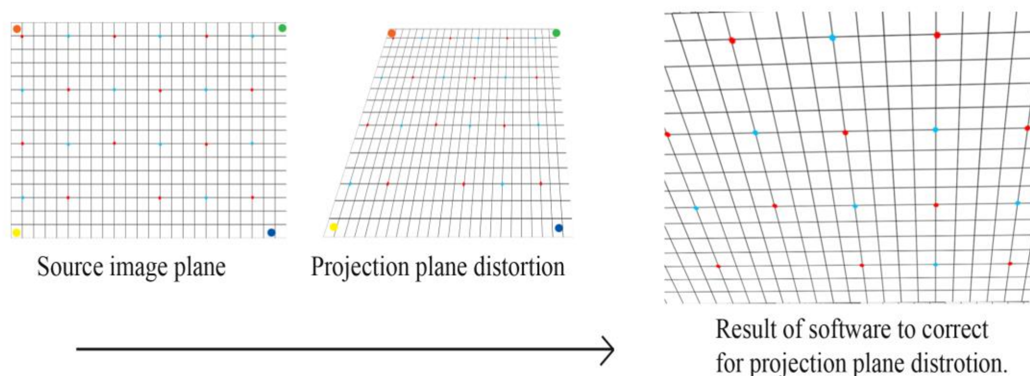



**Figure 4.4** shows the result of conduction a homographic transformation using our software. The second image is usedto show an example of a flat perspective distortion. The software calcualtes the difference between each of the image based on four points shown and produces a corretion so that the resulting image will appear undistorted on the projection surface.

We had little success with this strategy as well since it was only able to create a Mat based on a 3x3 matrix – which was not clearly documented, and, as before, only a homographic transformation (see figure 4.4). Experiments were preformed with different arrays of input points (see figure 4.5) and varying the matrix size in order to have more

degrees of freedom, but we were only ever able to achieve a homographic transformation. This then turned us back to the affine warping strategy.

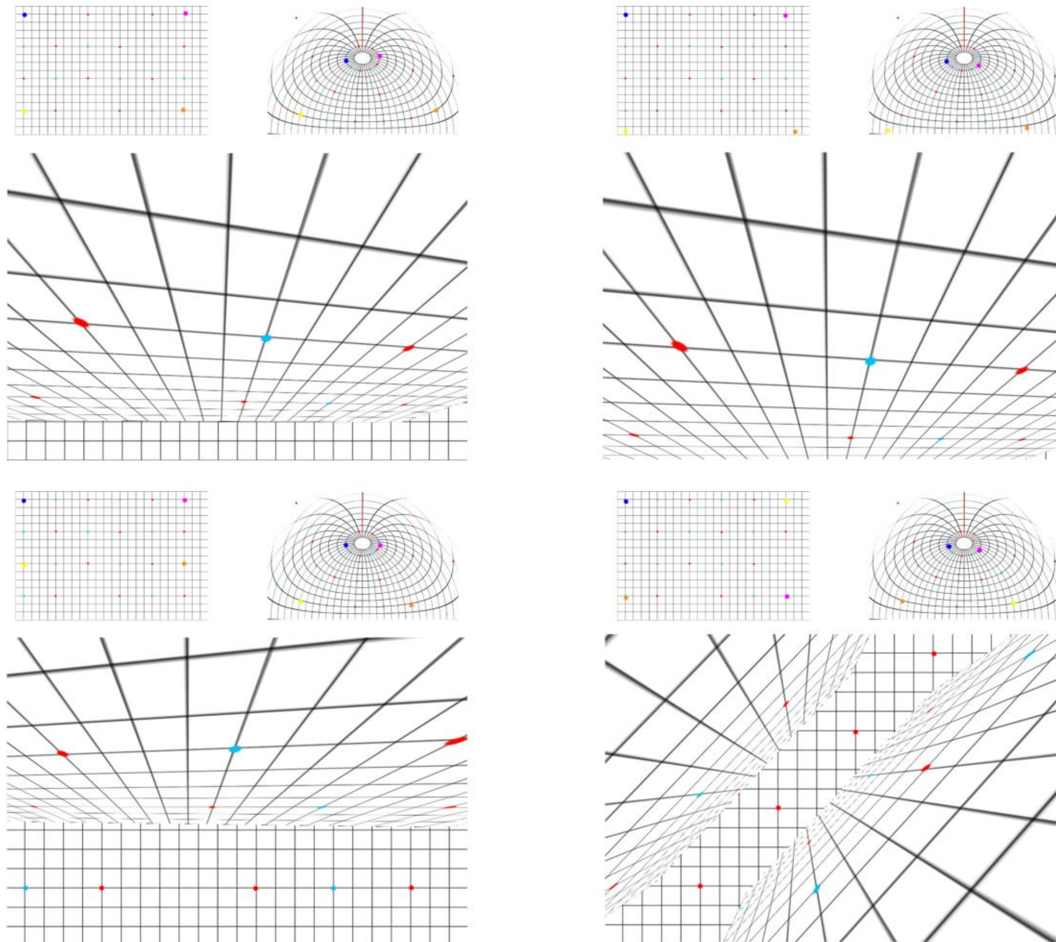


Figure 4.5. (top left) warp using the guide dots on each of the grids, and accouting for the vertical scaling distortion. (top right) warp using points as clsoe to the corners of each grid as possible. (bottom left) warp using the exact guide dot pattern. (bottom right) demonstration of the homography warp when the points are input into the array in non-corresponding order.

## 4.2.4 Piecewise Affine Warping

It seemed possible that we could manipulate this OpenCV based warping method to only warp certain parts of the source image, so once again if we could find a way to divide an image into the correct triangles, each triangle could be warped to fit a corresponding component of the dome. This was accomplished using the same

*getTransform()* method as before. As well as a similar strategy for capturing the mouse's position in a PVector to be taken as input for this method. The basic process behind piecewise affine warping here is

1. Split the image into corresponding triangles based on dome components, and on image contents, not vertex location - being sure to preserve their X and Y position.
2. Warp each triangle using a Mat derived from the triangle and a master warp image.
3. Re-render the full image out of each now-warped triangular piece of the image.

The main change from the previous software is the addition of a method called *alphaTriangle(Point[], PImage)*. This is used to separate out the triangle that exists based on the vertices of the triangle identified by *mouseclicked*. This is done by first calculating an alpha map of the image based on the given vertices with the Jama.solve method, which is passed a matrix of the three vertices in the form

$$A = \begin{bmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ 1 & 1 & 1 \end{bmatrix}$$

And a matrix created based on the x and y coordinates of each pixel in the returned PImage

$$b = \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

*A.solve(b)* (an equation of the form Ax=b) is solved for x, which is then given as a 3x1 matrix

$$x = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

Next we check if a, b and c are all positive coefficients, if they are it means that the pixel at (i, j) can be solved for using the vertices of our triangle and thus pixel (i, j) is inside of our triangle. Looping through all the pixels in the given PImage, alpha values of pixels in the triangle are set to 255 (completely opaque) and alpha values of pixels outside of the triangle are set to 0 (completely transparent). Note that these alpha values are only manipulated in the output PImage and nothing in the source image is actually modified. The output PImage is warped with the same method before and is then added to global list *PImage[] imgs.* The counters and arrays are then reset so the next triangle can be identified. Once all of the triangles are warped and added to the array a PGraphics object is rendered from each of the PImages in the array (each containing one warped triangle) and the final sketch is rendered for projection in the *draw()* function. The other option for rendering the image of combined triangles is to create one master PImage and or each image consisting of a single triangle transfer all of the pixels with alpha values of 255 to the master PImage and ignore all other pixels. Both options we explored (see figure 4.8).

## 4.3 Results and Final Projections

In order to calibrate our software to the specific distortions of the dome we projected a standard grid pattern with vertices marked every 4 units on the to dome surface. Then we marked where the field of view of the camera used to capture the images ended, figure 4.6 shows this, as well as an estimation of each triangle visible in the domes structure and an estimation of the borders of the projection surface visible through the camera.

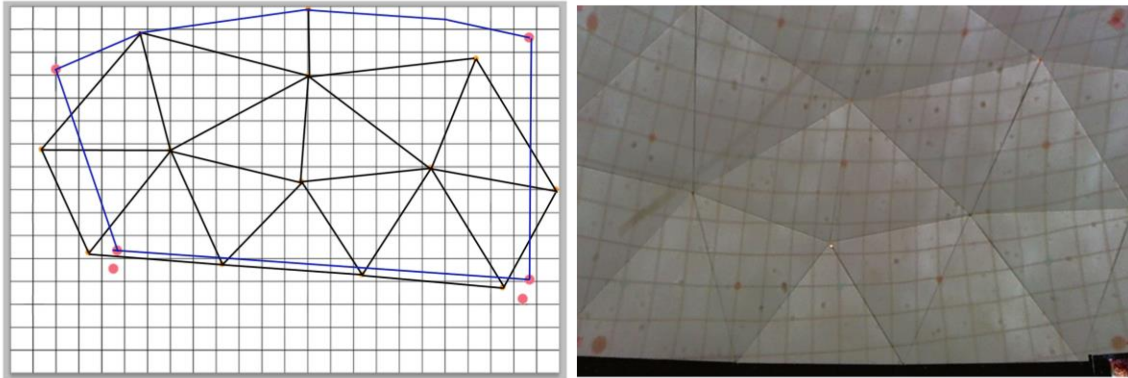


Figure 4.6 (right) the unmodified grid projected on the dome surface, each pick dot is marked on the grid image as to where a corner of the caturable field of view is. (left) the unwarped grid with triangles identified based on where each piece of the dom structure is located. the blue line shows the estimated full visible projection area.

Next we ran the un-warped grid image through the piecewise affine warping code, making sure that each triangle was correctly marked with the points on the grid image above. The marking of these triangles proved to be a difficult to control variable in the consistency of the projections (see figure 4.7).

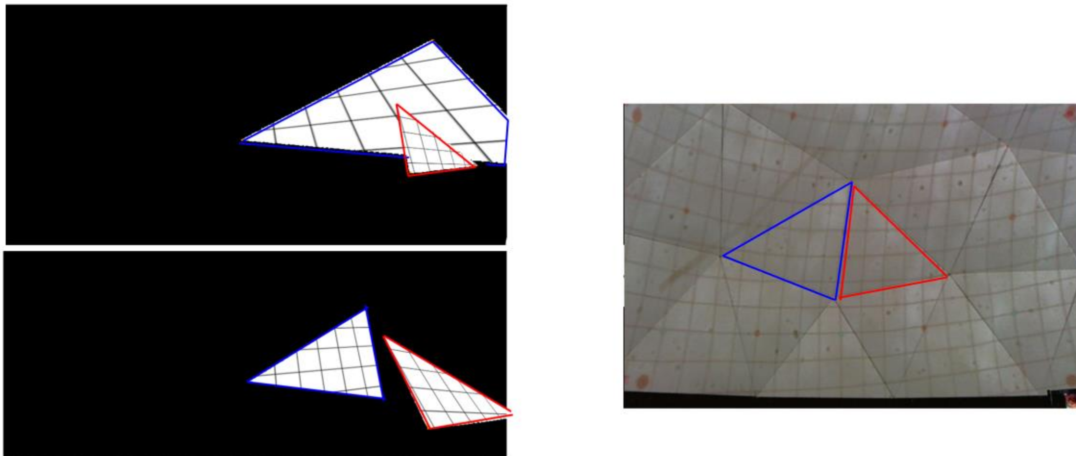


Figure 4.7. shows two diferent results of the piecewise affine warping software when tested on two triangels. The triangles are color coded. while the bottom result is more more desirable in terms of projection it would still not be considered a succeasful warp.

Another issue that arose in terms or projection consistency was the pros and cons of using different rendering methods. As discussed at the end of section 4.2 there are a few

different options when it comes to rendering each of the triangles into one draw-able PImage, PGraphics or a master PImage. PGraphics gives us the option of working from an array of unnamed PImages that are simply created from one image being manipulated each time the *alphaTriangle()* method is called. This however then presents the issues of more pointers and more memory required to process large images with many pieces. Creating a master PImage allows us to use the pixel array and transfer all of the pixels that are solved for based on the vertices of each triangle to the master PImage.

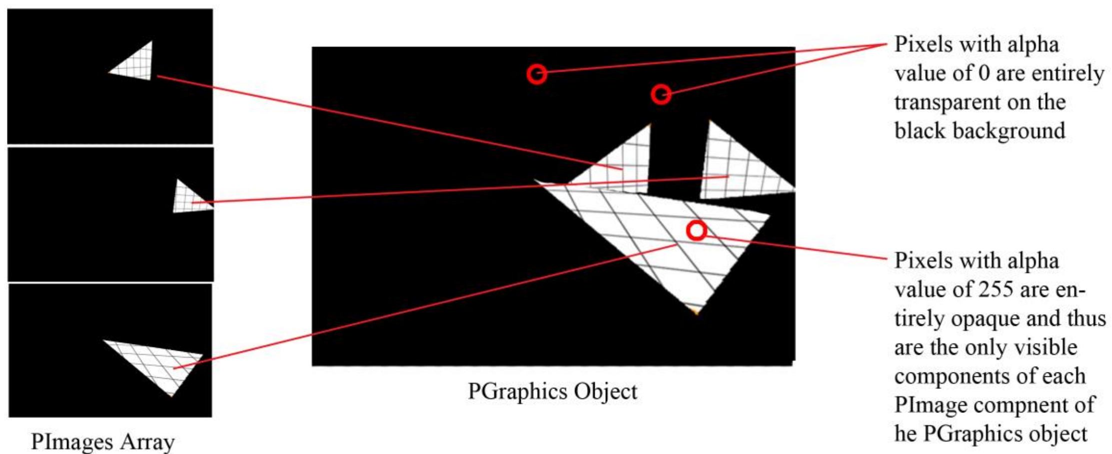



figure 4.8.1. shows the structure of how the PGrapics object is rendered from an input array of PImages. The pixels that are not in each of the triangles according to the Jama.solve method are completely transparent so that when the PImages are layered on top of each other only the opaque parts are seen.

After fine-tuning the method for identifying the triangles and their correspondences the final result of the project can be seen in figure 4.9 below. Note that there are slight discrepancies in the area of the triangles and gaps between in the final rendered image. These discrepancies are due to translating the pieces to the maser PImage only based on independent pixel arrays instead of implementing some method that would base the triangles position on the other triangles existing already in the master

PImage – thus making it so there would only be a need to anchor the first triangle translated into the master PImage.

The final change made to the piecewise affine warping program was to implement *getAffineTransform()* instead of the previous use of *getPerspectiveTransform()* in order to have the method take an input of 3 points (each of the three vertices of each triangle) to eliminate variation based on estimating the center point of the triangle as a fourth point.

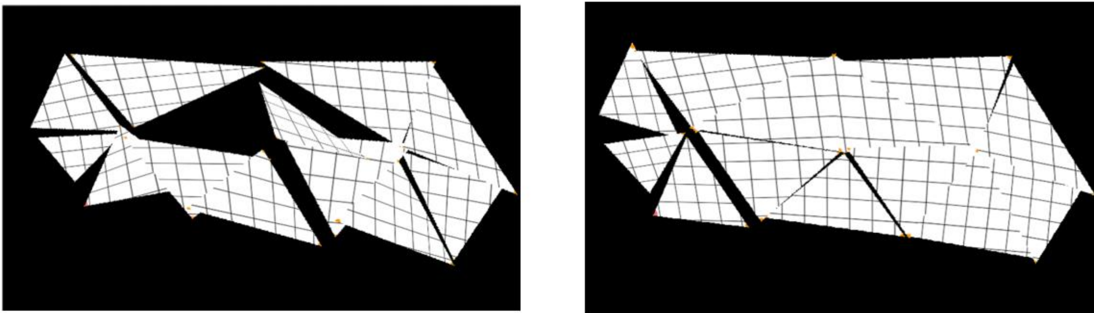
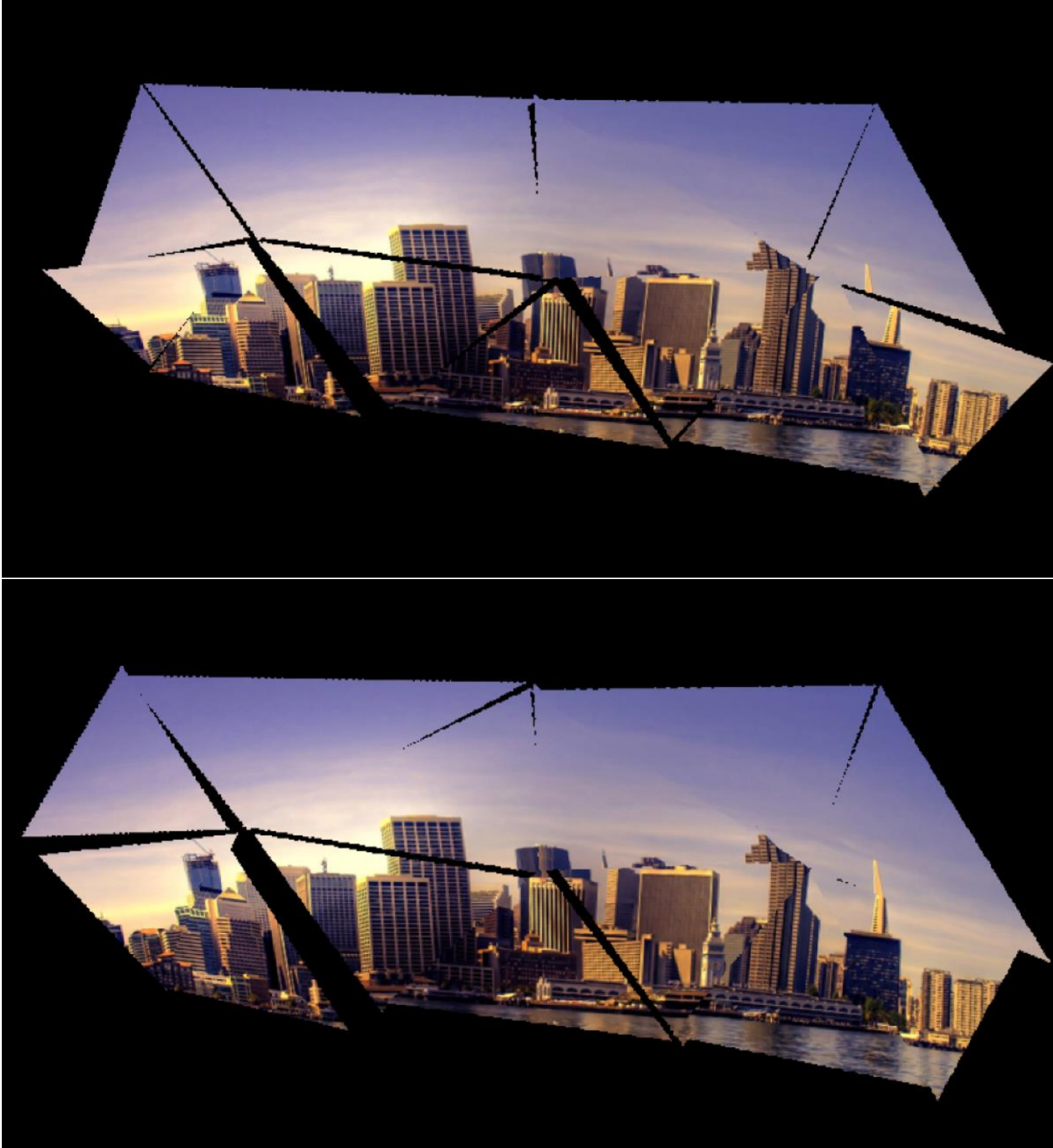
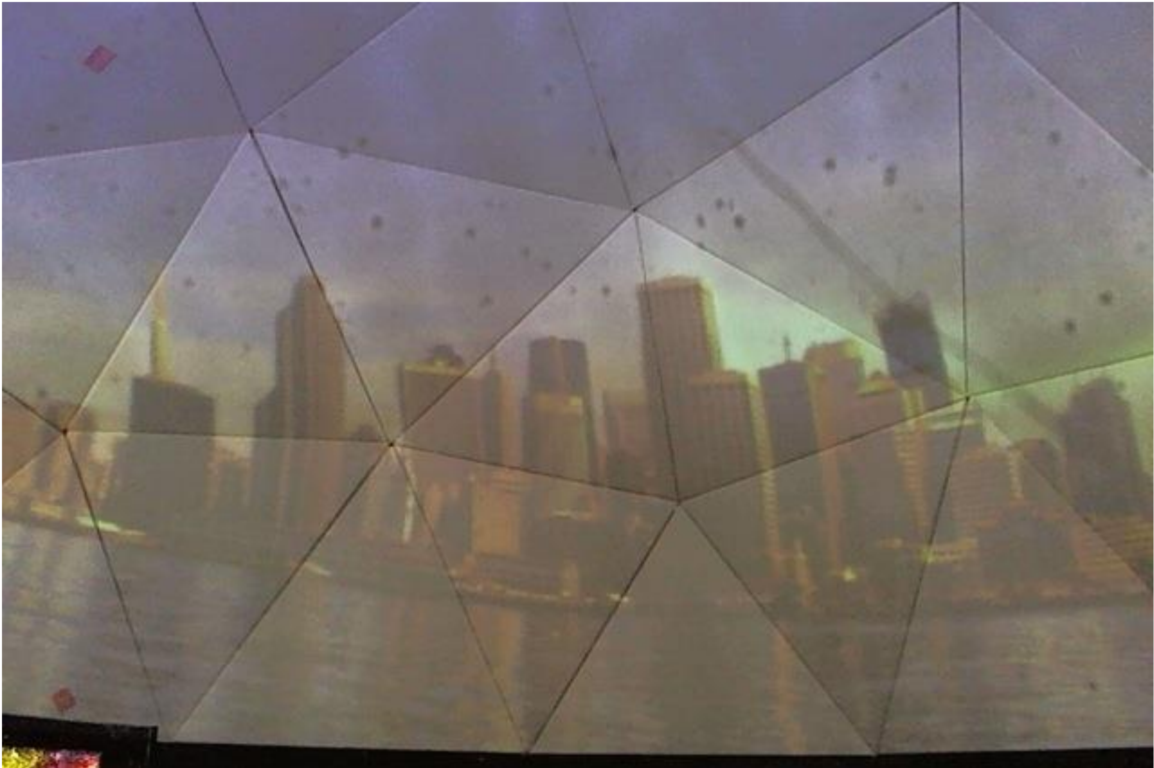


Figure 4.9.1 (left) a construction of the warped triangles using the warpPerspective and getPerspectiveTransform finctions in openCV and 4 points in each input array. (right) a construction of the triangles using warpAffine and getAffineTransform and 3 points in each input array. Note the corners in the affine warp construction are much tighter in relation to the neighborring triangles.

The original source image we used for the final warping and projection – chosen because it has a significant number of straight and parallel lines.



Above are two examples of a final render of the cityscape image after being run through the Piecewise affine warping software. There is a bit of distortion and inconsistency in selecting the exact vertices of each triangle that causes the black gaps to appear between each piece, at this point in the development of our software this is unavoidable except by precise clicking.



Above is the final projection (first an unmodified cityscape for comparison) using the cityscape image, some corrections still need to be made for tighter structure of the final rendered image as well as more exact alignment of the dome projection surface in relation to the projector face. However, it is noticeable that the perspective of the cityscape and the line are correct for each of the triangles – proving that the warping works. The shortcoming of the software occurs in the gaps in rendering.

# 5

# Conclusions and Future Work

## 5.1 Final Conclusions and analysis

We have presented 5 potential strategies for warping images to fit the distortion of a projection surface; spherical filters, piecewise affine warping, forward and backward pixel array-based warping, and warping using OpenCV algorithms. Each presents its own strengths and weaknesses, as discussed in detail earlier. For this project the most successful strategy for warping to fit the distortion of a flat, or 2D projection surface was the basic OpenCV warping. As for the dome and other 3D projection surfaces there are a few options. As discussed before spherical or fisheye filters are easy to implement but difficult to change if the projection surface is not a perfect sphere with the center of the projection on the center axis of the sphere. Forward and backward pixel array warping proved to have the same short comings, since the transformation had to be uniform across all pixels – based usually on distance form the center most pixel, it caused a perfect circular warp around which ever pixel is selected to be the root. The OpenCV strategies

proved to be the best for both 2D and 3D projection surfaces, using the basic algorithms in OpenCV we were able to easily correct for any 2D distortion. And using the initially proposed piecewise affine warping, implemented with OpenCV we were able to create software specifically tailored to our dome.

## 5.2 Future Work

The obvious next step after the work done in this project would be to alter the software we made for 3D warping to fit any given projection surface, such as one not constructed of triangles. Continuing this project we could also build software that would calculate the accuracy of potential different warping strategies as well as detect the edges and vertices of the image or piecewise triangles automatically.

To correct some of the errors in this project as it is, future research could build on our piecewise warping function and find a way of more accurately calculating the positioning of the triangles using either a 3D model of the dome with precise spatial measurements or by anchoring the projected triangles to the vertices of other triangles. There may also be a case to be made for redoing some of the research outlined here with more precise technology – a camera with a better FoV to capture more of the domes projection surface for example.

Research could be conducted with multiple projections and multiple projections building on work such as [11] using a two-view method for more accurate representation of the images in 3D spaces.

This project may also serve as a basis for other work surrounding projective transformation on images, with potential additions such as interaction with the projected environment, real time tracking that would shift the warp of the projected image based on the viewer's position, or converting the warping algorithm into a filter that could be applies to videos frame by frame for real time feed back to the dome. On the more technical side of this project, more work could be done on calculating the specific position of the camera in relation to the projection plane and the image plane when viewed on a computer screen as some of Paul Bourke's work details.

Lastly this project could be the starting point of building a media viewing application with different image filters and distortions based on the projection surface – it does not seem possible to render a video frame-by-frame to be projected into the dome, but it is possible that a video playing application could be built that would contain the warp designated by software like ours as a filter for the video to be played trough.

# References

[1] Azuma, Robert T. Survey of Augmented Reality, *Presence: Teleoperators & Virtual Environments, MIT Press Journal* (1997)

[2] Aggerwal, J.K. & Shishir Shah. Intrinsic Parameter Calibration Procedure for a (high-distortion) Fish-eye Lens Camera With Distortion Model and Accuracy Estimation, *Pattern Recognition* Vol. 29 No. 11 pp. 1775-1788 (1996)

[3] Vadaldi, Andrea, Brian Fulkerson. VLFeat – An Open and Protable Library of Computer Vision Algorithms, (2008)

[4] Bourke, Paul. Converting a Fisheye Image into a Panoramic, Spherical or Perspective Projection, (November 2004)

[5] Hartley, Richard, Sing Bing Kang. Parameter-Free Radial Distortion Correction with Center of Distortion Estimation, *IEEE Transactions of Pattern Analysis and Machine Intelligence,* Vol. 29, No. 8 (August 2007)

[6] Borenstein, Greg. OpenCV for Processing 0.5.2, https://github.com/atduskgreg/opencv-processing

[7] Weinbaum, Stanley Grauman, *Pygmalion's Spectacles,* 1935

[8] Davenport ET. AL. Synergistic Storyscapes and Constructionist Cinematic Sharing, *IMB Systems Journal,* Vol. 39 No's 3&4, (2000)

[9] Horowitz, Ken. Sega VR: Great Idea or Wishful Thinking, *Sega-16 Online Archive,* (December 2004)

[10] "Dome Projection." *Condeanticode.* Codeanticode, 07 September 2007, <https://codeanticode.wordpress.com/2013/09/06/dome-projection/>.

[11] Thi, Van Mai Nguyen. Development and Optimization of a Two-View Model for Anamorphic Projection on Planar Surfaces, *Bard College Division of Science, Mathematics and Computing* (May 2015)

[12] OpenCV Documentation, http://docs.opencv.org/2.4/index.html

[13] JAMA: A Java Matrix Package." *JAMA: Java Matrix Package*. N.p., n.d. Web. 29 Apr. 2017. <http://math.nist.gov/javanumerics/jama/>.

[14] Foundation, Processing. "Reference. Processing." *Processing Reference*. N.p., n.d. Web. 29 Apr. 2017. <https://processing.org/reference/>.

[15] Coonley, Ben. *Dreamlands: Immersive Cinema and Art 1905-2016.* Whitney Museum of American Art, New York, NY. http://bencoonley.com/

# Appendix A

# Processing Object Documentation [14]

PImage: Data type for storing images. Processing can display .gif, .jpg, .tga, and .png images. Images may be displayed in 2D and 3D space. Before an image is used, it must be loaded with the loadImage() function.
The PImage class contains fields for the width and height of the image, as well as an array called pixels[] that contains the values for every pixel in the image. The methods described below allow easy access to the image's pixels and alpha channel and simplify the process of compositing.

Before using the pixels[] array, be sure to use the loadPixels() method on the image to make sure that the pixel data is properly loaded.

To create a new image, use the createImage() function. Do not use the syntax new PImage().

PVector: A class to describe a two or three dimensional vector, specifically a Euclidean (also known as geometric) vector. A vector is an entity that has both magnitude and direction. The datatype, however, stores the components of the vector (x,y for 2D, and x,y,z for 3D). The magnitude and direction can be accessed via the methods mag() and heading().

In many of the Processing examples, you will see PVector used to describe a position, velocity, or acceleration. For example, if you consider a rectangle moving across the screen, at any given instant it has a position (a vector that points from the origin to its location), a velocity (the rate at which the object's position changes per time unit, expressed as a vector), and acceleration (the rate at which the object's velocity changes per time unit, expressed as a vector). Since vectors represent groupings of values, we cannot simply use traditional addition/multiplication/etc. Instead, we'll need to do some "vector" math, which is made easy by the methods inside the PVector class.

Pgraphics: Main graphics and rendering context, as well as the base API implementation for processing "core". Use this class if you need to draw into an off-screen graphics buffer. A PGraphics object can be constructed with the createGraphics() function. The beginDraw() and endDraw() methods (see above example) are necessary to set up the buffer and to finalize it. The fields and methods for this class are extensive. For a complete list, visit the developer's reference.

To create a new graphics context, use the createGraphics() function. Do not use the syntax new PGraphics().

# Appendix B
# Java Code

```
1.  OpenCV_Warp
2.  
3.  import gab.opencv.*;
4.  import org.opencv.imgproc.Imgproc;
5.  import org.opencv.core.MatOfPoint2f;
6.  import org.opencv.core.Point;
7.  import org.opencv.core.Size;
8.  
9.  import org.opencv.core.Mat;
10. import org.opencv.core.CvType;
11. 
12. import Jama.*;
13. import processing.video.*;
14. 
15. //create the two opencv objects we will use for the images
16. OpenCV opencvSrc, opencvWarp;
17. // create the PImage used to assign to the Opencv objects and for the final render
18. PImage src, warp, dest;
19. 
20. //set parameters of how many points we need to close the arrays
21. int np1 = 0;
22. int np2 = 0;
23. int imgNum = 0;
24. 
25. //create the arrays that will be given to getTransformation as input
26. ArrayList<PVector> srcArray = new ArrayList<PVector>();
27. ArrayList<PVector> dstArray = new ArrayList<PVector>();
28. 
29. //define our Mat object
30. Mat warpMat;
31. 
32. void setup() {
33.   src = loadImage("Grid.jpg"); //unwarped grid
34.   size(src.width * 2, src.height);
35.   //establish src as an opencv object
36.   opencvSrc = new OpenCV(this, src);
37.   opencvSrc.blur(1);
38.   opencvSrc.threshold(120);
39. 
40.   warp = loadImage("warpedgrid.jpg"); //distorted grid that we want to find the
    transformation of
41.   opencvWarp = new OpenCV(this, warp);
42.   opencvWarp.blur(1);
43.   opencvWarp.threshold(120);
44. }
45. 
46. //using the mouse cursor, fill each of the PVector arrays with 4 cooresponding points
    from each image
47. void mousePressed() {
48.   int i = int(mouseX / (src.width));
49.   //check to see if the cursor is in the left image (src)
50.   if (i == 0) {
51.     //create a PVector of the mouse location in the form (mouseX, mouseY, 1)
52.     srcArray.add(np1, new PVector(mouseX % (src.width), mouseY % (src.height), 1));
53.     np1++;
```

```
54.     //print out the loaction so we have conformation the array is being popualted
55.     println("pic 1" + srcArray.get(np1-1));
56.     //check to see if cursor is in right image (warp)
57.   } else if (i == 1) {
58.     dstArray.add(np2, new PVector(mouseX % (src.width), mouseY % (src.height), 1));
59.     np2++;
60.     println("pic 2" + dstArray.get(np2-1));
61.   }
62.   //if each array is populated by at least 4 PVector points initiate the matrix
    estimation
63.   if (np1 >=4 && np2 >= 4 && np1 == np2) {
64.     warpMat = warpPerspective(srcArray, dstArray, src.width, src.height);
65.   }
66. }
67.
68. //using the populated PVector arrays and built in OpenCV finctions, to calculate the
    transformation
69. //between the two images the points are sourced form
70. Mat getTransform(ArrayList<PVector> srcArray, ArrayList<PVector> dstArray) {
71.   Point[] srcPoints = new Point[4];
72.   Point[] dstPoints = new Point[4];
73.   //convert each PVector ArrayList to an array of Points (a intrinsic prcessing object)
74.   for (int i = 0; i <4; i++) {
75.     srcPoints[i] = new Point(srcArray.get(i).x, srcArray.get(i).y);
76.   }
77.
78.   for (int i = 0; i <4; i++) {
79.     dstPoints[i] = new Point(dstArray.get(i).x, dstArray.get(i).y);
80.   }
81.   //create a MatOfPoint2f object from each of the point array
82.   MatOfPoint2f srcMarker = new MatOfPoint2f();
83.   srcMarker.fromArray(srcPoints);
84.
85.   MatOfPoint2f dstMarker = new MatOfPoint2f();
86.   dstMarker.fromArray(dstPoints);
87.   //using the Image Processing module in OpenCV build the Mat object using a built in
88.   //function getPerspetiveTransform.
89.   return Imgproc.getPerspectiveTransform(dstMarker, srcMarker);
90. }
91.
92. //applies the transformation Mat calculated above to the Opencv object created for src
93. Mat warpPerspective(ArrayList<PVector> srcArray, ArrayList<PVector> dstArray, int w, int
    h) {
94.   Mat transform = getTransform(srcArray, dstArray);
95.   Mat unWarpedMarker = new Mat(w, h, CvType.CV_8UC1);
96.   Imgproc.warpPerspective(opencvSrc.getColor(), unWarpedMarker, transform, new Size(w,
    h));
97.   return unWarpedMarker;
98. }
99.
100.          void draw() {
101.            //if the PVector arrays are not yet full
102.            if (np1 < 4 || np2 < 4) {
103.              image(src, 0, 0);
104.              noFill();
105.              stroke(0, 255, 0);
106.              strokeWeight(4);
107.              translate(src.width, 0);
108.              image(warp, 0, 0);
109.              //once arryas are full create a new PImgae and convert the Mat of src and
    the transfomration matrix to that PImage.
```

```
110.           } else if (np1 >= 4 && np2 >= 4 && np1 == np2) {
111.             dest = createImage(src.width, src.height, ARGB);
112.             opencvSrc.toPImage(warpMat, dest);
113.             image(dest, 0, 0);
114.           }
115.         }
```

```
1.  //Triangle Warp
2.  
3.  import gab.opencv.*;
4.  import org.opencv.imgproc.Imgproc;
5.  import org.opencv.core.MatOfPoint2f;
6.  import org.opencv.core.Point;
7.  import org.opencv.core.Size;
8.  
9.  import org.opencv.core.Mat;
10. import org.opencv.core.CvType;
11. 
12. import Jama.*;
13. import processing.video.*;
14. 
15. OpenCV opencvSrc, opencvWarp, opencvTri;
16. PImage src, warp, tri, destTemp, dest, real;
17. int MAXIMGS = 13; //number of tirangles in projection surface
18. 
19. PGraphics output; //another option for final render
20. 
21. PImage[] imgs = new PImage[MAXIMGS];
22. int np1 = 0; //number of points in image 1
23. int np2 = 0; //number of points in image 2
24. int imgNum = 0; //number of warped pieces
25. 
26. ArrayList<PVector> srcArray = new ArrayList<PVector>();
27. ArrayList<PVector> dstArray = new ArrayList<PVector>();
28. 
29. Mat warpMat;
30. 
31. Point[] srcPoints = new Point[3];
32. Point[] dstPoints = new Point[3];
33. 
34. void setup() {
35.   real = loadImage("real.jpg");
36. 
37.   src = loadImage("grid layer.jpg");
38.   size(src.width * 2, src.height);
39.   opencvSrc = new OpenCV(this, src);
40.   opencvSrc.blur(1);
41.   opencvSrc.threshold(120);
42. 
43.   warp = loadImage("realgrid.jpg");
44.   opencvWarp = new OpenCV(this, warp);
45.   opencvWarp.blur(1);
46.   opencvWarp.threshold(120);
47. 
48.   output = createGraphics(src.width, src.height, JAVA2D);
49. 
50.   destTemp = createImage(src.width, src.height, RGB);
51.   dest = createImage(src.width, src.height, RGB);
52. }
53. 
```

```
54. void mousePressed() {
55.   int i = int(mouseX / (src.width));
56.   if (i == 0) {
57.     srcArray.add(np1, new PVector(mouseX % (src.width), mouseY % (src.height), 1));
58.     np1++;
59.     println("pic 1" + srcArray.get(np1-1));
60.   } else if (i == 1) {
61.     dstArray.add(np2, new PVector(mouseX % (src.width), mouseY % (src.height), 1));
62.     np2++;
63.     println("pic 2" + dstArray.get(np2-1));
64.   }
65.   if (np1 >=3 && np2 >= 3 && np1 == np2) {
66.     toPointArray(srcArray, dstArray);
67.     alphaTriangle(srcPoints, src);
68.     warpMat = warpPerspective(srcArray, dstArray, src.width, src.height);
69.     opencvTri.toPImage(warpMat, destTemp);
70.     renderPixel(destTemp, dest);
71.     imgs[imgNum] = dest;
72.     imgNum++;
73.     println(imgNum);
74.     np1 = 0;
75.     np2 = 0;
76.     println("reset" + imgNum);
77.   }
78. }
79.
80. void alphaTriangle(Point[] points, PImage img) {
81.   /*loop through the pixels in our image and based upon the three
82.    vertices selected for each triangle, calculate whether each pixel
83.    exists in the triangle defined by three selected ponts, if so set alpha
84.    value to max, if not set alpha value to 0. */
85.
86.
87.   tri = createImage(img.width, img.width, ARGB);
88.
89.   Matrix alphaMat;
90.
91.   double[][] array = {
92.     {
93.       points[0].x, points[1].x, points[2].x
94.     }
95.     , {
96.       points[0].y, points[1].y, points[2].y
97.     }
98.     , {
99.       1, 1, 1
100.            }
101.          };
102.          Matrix pointMat = new Matrix(array);
103.          pointMat.print(3, 3);
104.
105.          img.loadPixels();
106.          tri.loadPixels();
107.
108.          for (int y = 0; y < img.height; y++) {
109.            for (int x = 0; x < img.width; x++) {
110.              int loc = x + y*img.width;
111.
112.              double[] tempArray = {
113.                x, y, 1
114.              };
```

```
115.
116.                Matrix p = new Matrix(tempArray, 3);
117.
118.                float r = red(real.pixels[loc]);
119.                float g = green(real.pixels[loc]);
120.                float b = blue(real.pixels[loc]);
121.
122.                alphaMat = pointMat.solve(p);
123.
124.                if (alphaMat.get(0, 0) >= 0 && alphaMat.get(1, 0) >= 0 && alphaMat.get(2,
   0) >= 0) {
125.                  tri.pixels[loc] = color(r, g, b, 255);
126.                } else {
127.                  tri.pixels[loc] = color(r, g, b, 0);
128.                }
129.              }
130.            }
131.            tri.updatePixels();
132.            opencvTri = new  OpenCV(this, tri);
133.          }
134.
135.          void toPointArray(ArrayList<PVector> srcArray, ArrayList<PVector> dstArray) {
136.            //converts the PVector arrays created by mousepressed to Point object arrays
137.
138.
139.            for (int i = 0; i <3; i++) {
140.              srcPoints[i] = new Point(srcArray.get(i).x, srcArray.get(i).y);
141.            }
142.
143.            for (int i = 0; i <3; i++) {
144.              dstPoints[i] = new Point(dstArray.get(i).x, dstArray.get(i).y);
145.            }
146.          }
147.
148.
149.          Mat getTransform(ArrayList<PVector> srcArray, ArrayList<PVector> dstArray) {
150.            //same as OpenCV_warp except uisng arrays of 3 Points and
151.            //Affine transformation matrices instead
152.
153.
154.            MatOfPoint2f srcMarker = new MatOfPoint2f();
155.            srcMarker.fromArray(srcPoints);
156.
157.            MatOfPoint2f dstMarker = new MatOfPoint2f();
158.            dstMarker.fromArray(dstPoints);
159.
160.            return Imgproc.getAffineTransform(dstMarker, srcMarker);
161.          }
162.
163.          Mat warpPerspective(ArrayList<PVector> srcArray, ArrayList<PVector> dstArray,
   int w, int h) {
164.            Mat transform = getTransform(srcArray, dstArray);
165.            Mat unWarpedMarker = new Mat(w, h, CvType.CV_8UC1);
166.            Imgproc.warpAffine(opencvTri.getColor(), unWarpedMarker, transform, new
   Size(w, h));
167.            return unWarpedMarker;
168.          }
169.          }
```

# Appendix C
# Other Code Resources

imgProc: http://docs.opencv.org/2.4/modules/imgproc/doc/imgproc.html
OpenCV's image processing package – includes functions: warpPerspective, getPerspectiveTransform & warpAffine

JAMA: A basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real, dense matrices. It is meant to provide sufficient functionality for routine problems, packaged in a way that is natural and understandable to non-experts.

JAMA is comprised of six Java classes: Matrix, CholeskyDecomposition, LUDecomposition, QRDecomposition, SingularValueDecomposition and EigenvalueDecomposition.

The Matrix class provides the fundamental operations of numerical linear algebra. Various constructors create Matrices from two dimensional arrays of double precision floating point numbers. Various *gets* and *sets* provide access to submatrices and matrix elements. The basic arithmetic operations include matrix addition and multiplication, matrix norms and selected element-by-element array operations. A convenient matrix print method is also included.

Five fundamental matrix decompositions, which consist of pairs or triples of matrices, permutation vectors, and the like, produce results in five decomposition classes. These decompositions are accessed by the Matrix class to compute solutions of simultaneous linear equations, determinants, inverses and other matrix functions. The five decompositions are

- Cholesky Decomposition of symmetric, positive definite matrices
- LU Decomposition (Gaussian elimination) of rectangular matrices
- QR Decomposition of rectangular matrices
- Eigenvalue Decomposition of both symmetric and nonsymmetric square matrices
- Singular Value Decomposition of rectangular matrices

The current JAMA deals only with real matrices. We expect that future versions will also address complex matrices. This has been deferred since crucial design decisions cannot be made until certain issues regarding the implementation of complex in the Java language are resolved. [13]

Professor O'Hara's Computer Vision Assignment - http://drablab.org/keithohara/cmsc-317-2014f/assignments/cmsc317assignment6.html