3-2017

Mapping Images onto Solids in Mathematica

Zachary Ash
Grand Valley State University, ashz@mail.gvsu.edu

Mapping Images onto Solids in *Mathematica*

Zachary Ash

Advisor: Dr. Edward Aboufadel

Grand Valley State University

March 2017

Contact information: aboufade@gvsu.edu

Abstract

The goal of this research is to design a flexible method for mapping a two-dimensional grayscale image onto the surface of a three-dimensional solid. The approach used should be relatively easy to adapt to various solids without redesigning the entire process as well as able to map the entire image onto the entire object – partial coverage of the object and partial usage of the image are to be avoided. Having decided on an approach, the method is then to be designed in *Mathematica* to produce an STL file of the object with the desired grayscale image embossed or engraved onto it. This STL file can then be used to create a 3D print.

This research will draw on college-level mathematics, especially such areas as calculus and projective geometry which deal more with three-dimensional surfaces and modeling. Some elements of differential calculus will also prove to be useful in terms of determining normal vectors and other features which can be useful in determining an appropriate mapping. This research also focuses on algorithms very similar to those used in *uv*-mapping and bump mapping, which are methods often used in computer graphics to map textures and/or three-dimensional "bumps" onto a solid without actually including these details in the raw model. Both of these are variations of texture mapping, the process of adding detail to a model by mapping some image onto it. It can also be thought of the reverse version of map projections – rather than converting a solid into an image representing it, the image is molded into the surface of a solid.

Overall, the primary method that is developed in this paper yields consistent results, with minimal distortion on low curvature surfaces. The distortion introduced into the image depends heavily on the surface being mapped onto, since surfaces like that of a sphere will tend to generate significant distortions toward each pole.

Introduction

In order to define methods to map an image onto a solid, we must first define the various elements dealt with in this paper. Descriptions of both grayscale images and STL files are included here for better understanding of the purpose of this work.

A grayscale image is a digital image where each pixel (i.e. point) contains only the intensity of the image, that is to say there is no color information, only a single value. This is often visualized as varying from black (0) to white (the max value) with shades of gray in between, hence the name “grayscale” [Christensen]. Thus, when processed by a program, a grayscale image is identical in form to a two-dimensional array of numerical values (often 0 – 255 for integer values or 0.0 – 1.0 for floating point values). In terms of mapping such an image onto a solid, these values will indicate the amount the surface of the solid should be embossed or engraved at the corresponding point.

Figure 1: An Example of a Grayscale Image (Grand Valley State Lakers Logo).

An STL (STereoLithography) file is a file for storing three-dimensional objects in a digital format. The essence of the formatting is that each solid’s surfaces are broken down into a triangulation, and then each triangle is stored to model the original object. Each triangle (a “facet”) is stored as a collection of three vertices and a normal vector. Note in the image below that many “facets” will be included inside the definition of a single “solid” (the entire object). While the format shown below is defined in terms of text, there is also a binary format for more condensed files [Burns].

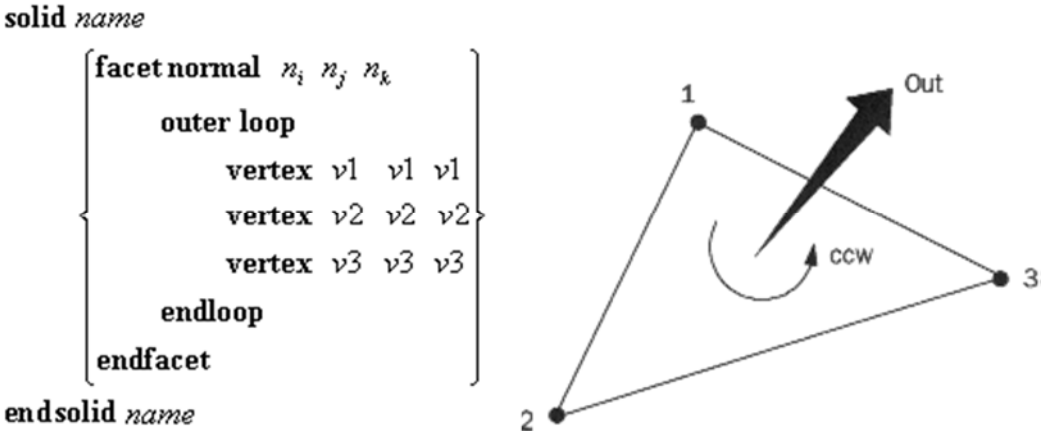

Figure 2: The Representation of a Triangle in STL Format (Image Source: STL).

STL files are the format of choice for many products involving three-dimensional models, such as 3D printers, so the format is widely offered by CAD and other modeling software. Since the goal of this work is to print our models on a 3D printer, the built-in converter in *Mathematica* will be used to generate STL versions of the generated objects.

The general concept of any algorithm to be used to map a two-dimensional image to a three-dimensional solid is composed of a few common elements. All of the algorithms will take some representation of a 3D object (function, STL, point list…) and a 2D image (2D array, function…). The algorithms will then by some means map each grayscale value in the image to a point on the solid (or map each point on the solid to a spot on the image) – this is the primary challenge of mapping an image onto a complex solid. Finally, the algorithm will use the image values to emboss or engrave the solid, and will output some representation of the modified 3D object.

The objects that will be generated through these algorithms may be manipulated and converted to STL files in *Mathematica*. *Mathematica* also provides the tools to load images as arrays and convert them into useful forms for processing them. The STL files generated may then be transferred to a 3D printer for actually printing the model [Aboufadel].

Procedure

Three potential algorithms to map an image onto a solid are discussed:

1) The "Mapping-Function" Method:

This method requires that for any solid, a function be given that can map every point of the image onto the desired solid. In other words, this approach uses a function $f: [0,1]^2 \times \mathbb{R} \rightarrow \mathbb{R}^3$. Then, $f(\{x_i, y_i, g_i\}) = \{x_o, y_o, z_o\}$ for any $x_i, y_i \in [0,1]$ and $g_i, x_o, y_o, z_o \in \mathbb{R}$ such that (x_i, y_i) is a point in the image's coordinate-frame where (0, 0) is the bottom left corner of the image and (1, 1) is the top right corner, g_i is the grayscale value of the image at (x_i, y_i), and (x_o , y_o , z_o) is the resulting point on the solid (properly embossed or engraved).

This process can be somewhat simplified for the user by reducing the function to $f: [0,1]^2 \rightarrow \mathbb{R}^3$, where the inputs and outputs now are: $f(\{x_i, y_i\}) = \{x_o, y_o, z_o\}$, where ($x_o$, y_o , z_o) is no longer the embossed or engraved point, but merely the unchanged point on the original solid. By removing the grayscale input from the function, the user then only has to define the function to generate the points on the solid. Once the function to generate the "normal" solid has been defined, the algorithm will generate all the points and use the normal vectors at each point and the corresponding image grayscale value to determine how to emboss or engrave the solid.

Having designed the function to generate the points on the solid, the output from this step will be a point cloud of points on the solid's surface, i.e. it will generate the list:

$$\{p_{11}, p_{21}, \dots p_{W1}, p_{12}, p_{22}, \dots p_{1H}, p_{2H}, \dots p_{LH}\}$$

where $p_{i,j}$ is the point corresponding to pixel (i, j) in the two-dimensional image. Thus, for the width of the image, W, and the height of the image, H, (in pixels) i and j are any natural numbers such that $1 \leq i \leq W, 1 \leq j \leq H$. Note that for any i and j, $\left(x_{input}, y_{input}\right) = \left(\frac{i}{W}, \frac{j}{H}\right)$ is the input to f.

In order to create the actual STL output, the points generated in the last step must still be connected together to form triangles on the surface of the solid. In other words, from the point cloud created above, an actual solid must still be built. By forming a list of collections of three points (where the three points are the vertices of a triangle on the solid), a mesh can be formed from the point cloud and thus an STL can be created. Thus, given a list of the points outputted by the previous step in the form $\{p_1, p_2, p_3, \dots p_n\}$, the output of this step will be of the form:

$$\{\{i_{11}, i_{12}, i_{13}\}, \{i_{21}, i_{22}, i_{23}\}, \dots \{i_{m1}, i_{m2}, i_{m3}\}\},$$

for any $i_{l,k}$ such that $p_{i_{l,k}}$ is the k^{th} point in the i^{th} triangle on the solid's surface, where m is the number of triangles on the solid and k and j are any natural numbers such that $1 \leq k \leq 3$,

$1 \leq l \leq m$. Then, the final STL can be formed as a 3D Mesh created from the point cloud and the triangle indices generated in these steps.

Since this approach requires that a function be given to generate the solid's points based on the image values, it is only feasible to use common solids such as cylinders, cubes, or spheres. These solids are relatively easy to generate from functions. Granted, any shape can be generated with a complicated enough function to a small enough resolution, but this is far from an ideal approach to any shapes without much symmetry or well-defined surfaces. For example, a torus is a reasonably complex surface that can be generated via this method (thanks to its symmetry), but any arbitrary shape, such as a teapot, can be incredibly difficult or tedious to define and generate

2) The "Vacuum-Fit" Method:

This method builds off the first method. The mapping function described above is used to map an image onto some simple solid (e.g. a sphere). Then, this simple solid is scaled to enclose the entire final solid. By shrinking the simple model until it contacts the complex model, points from the image on the simple model can be assigned to those on the complex model. By fixing those points, the simple model can then be tightened down onto the complex model more thoroughly until all the points from the image have been associated with a point on the complex solid's surface. Then, having mapped the image points onto the solid's surface, the only thing that remains is to emboss or engrave the solid's surface the appropriate amount along its normal vector.

This method can also be used "backwards" by starting with a small version of the simple solid inside the complex solid and "inflating" it until it contacts the inside surface of the solid. Then, as with the first variation, the points from the image are associated with the contacting point of the solid and used to emboss or engrave the solid.

This method is relatively flexible, since it does not require the user to define a function for every shape that an image is to be mapped onto. It also provides some additional flexibility with how to map the image onto the solid by allowing different "intermediate simple solids" to be used which dictate how the image will be scaled and distributed on the final solid.

3) The "Unraveling the Triangles" Method:

This method is completely distinct from the other two methods, in that instead of manipulating the image to a desired shape or molding it onto the solid, it relies on taking the final solid and flattening it onto the two-dimensional image. By starting with a triangulated solid, this method then "cuts" and "unfolds" the triangles of the model repeatedly until it can be flattened onto the image. By placing this flat model onto the image, a correspondence from each point in the image to the points on the model can be found and used to determine the grayscale value of the image at each point. By returning the model to its original shape, these

values can be then used to translate each point along its normal vector the appropriate distance as determined by the grayscale value associated with it.

This approach, while very versatile, is much more complex than the other methods described. Writing a program to follow these steps is challenging, and tends to yield unpredictable and inconsistent results based on how the "cuts" and "folds" are picked on a given model. Without human supervision of the process, this approach can become very messy, and the flattened models tend to not use the entirety of the image provided, which can result in parts of the image not being mapped onto the solid at all.

Due to the complexity of the third approach and the dependence of the second method upon the first, they will not be addressed any further in this paper. The first method, however, will both be discussed in further detail.

In order to create models of each solid to be used in this work, a function must be designed (as described above) such that the position of the given pixel and the value of the image at that point are mapped to a point on the solid. Note that each individual solid includes extra constants for the purpose of scaling some aspect of the solid (e.g. height, radius). The code for generating the solids discussed below (with the image mapped onto it) is presented in Appendix A. Note that this code works for both embossing and engraving images onto the solids – using images with negative values will engrave the surface, whereas using images with positive values will emboss the surface.

Cylinder

In order to map the image onto a cylinder, the image can be "wrapped" around the curved surface of the cylinder such that the left and right sides meet. Extra points must then be added to complete the top and bottom of the cylinder. Given the radius *r* and the height *h* of a cylinder, the function to generate the solid is

$$f(x_i, y_i, g_i) = ((r + g_i) * \cos(2\pi x_i), (r + g_i) * \sin(2\pi x_i), h * y_i)$$

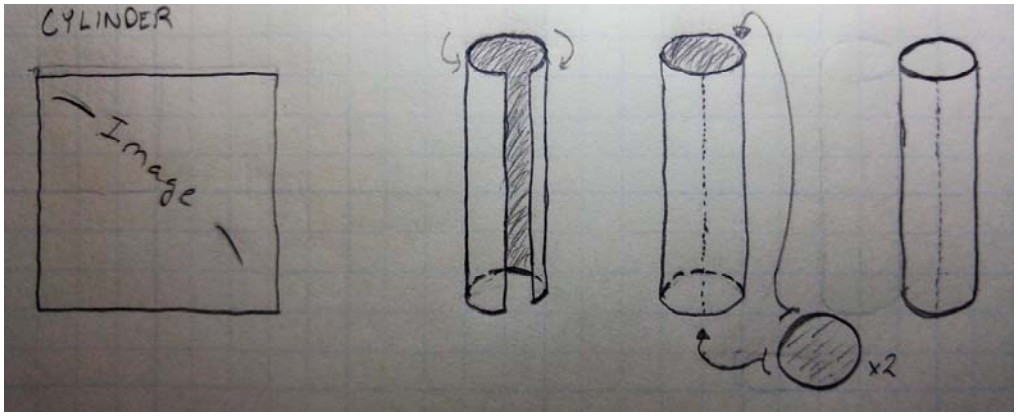

Figure 3: Wrapping an image onto a cylinder.

Sphere

Generating a sphere is similar to the process of a cylinder, but as the points approach either pole, the radius is "tightened" so that the top and bottom converge to a single point. Given the radius *r*, the sphere can be generated by this function:

$$f(x_i, y_i, g_i) = ((r + g_i) * \cos(2\pi x_i)\sin(\pi y_i), (r + g_i) * \sin(2\pi \mathrm{x_i})\sin(\pi y_i), (r + g_i) * \cos(\pi y_i))$$

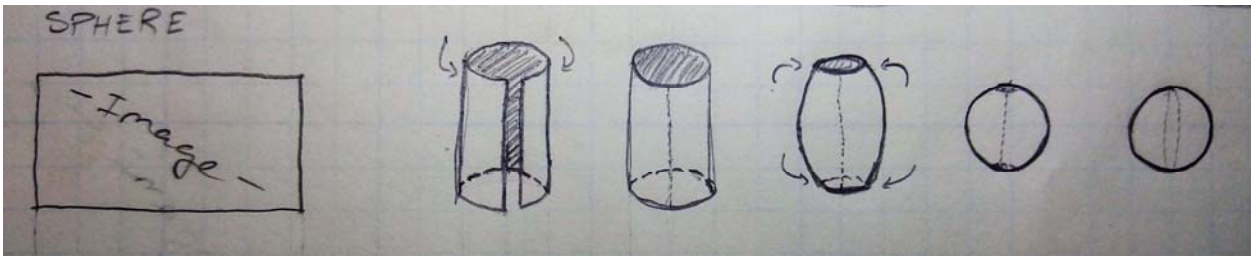

Figure 4: Wrapping an image onto a sphere.

Cube

The cube is harder to generate, given its definition is more of a piecewise function to generate each face. For simplicity, the image may be mapped to only four sides, thus leaving the top and bottom to be added later (as with the cylinder). Then, by dividing the image into four equal sections, each section may be mapped onto the plane of one face of the cube (which is just a matter of scaling). Given the side length *s*, the cube's piecewise function is given:

$$f(x_i, y_i, g_i) = \begin{cases} (-4sx_i, -g_i, sy_i), & 0 \le x_i < 0.25 \\ (-(s + g_i), 4s(x_i - 0.25), sy_i), & 0.25 \le x_i < 0.5 \\ (-4s(0.75 - x_i), s + g_i, sy_i), & 0.5 \le x_i < 0.75 \\ (g_i, 4s(1 - x_i), sy_i), & 0.75 \le x_i \le 1 \end{cases}$$

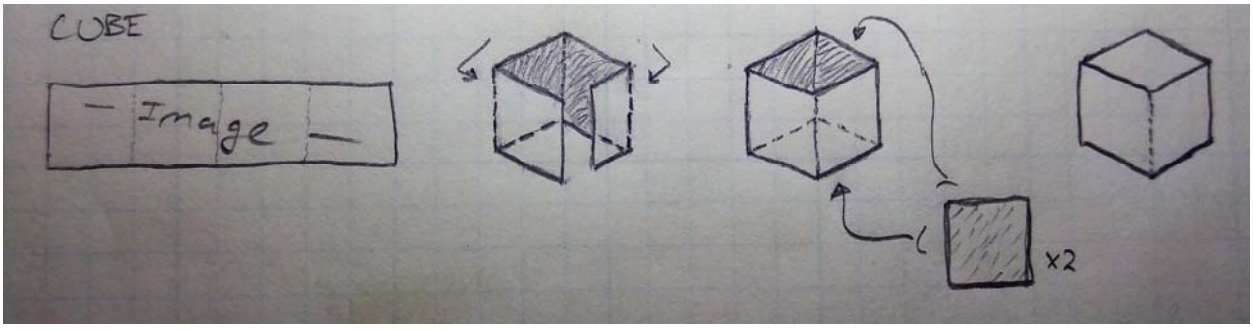

Figure 5: Wrapping an image onto a cube.

Torus

Mapping an image to a torus is similar to that of a cylinder, but after the initial wrap, the top and bottom loops are then folded so that they also touch. Thus, the cylinder's equations can be seen in the parts involving the smaller radius of the torus, but then the larger radius also affects those components. Given the "little" radius *r* and the "large" radius *R*, the torus is generated by the function

$$f(x_i, y_i, g_i) = ((R + (r + g_i)\cos(2\pi y_i)) * \cos(2\pi x_i), (R + (r + g_i)\cos(2\pi y_i)) * \sin(2\pi x_i), (r + g_i) * \sin(2\pi y_i))$$

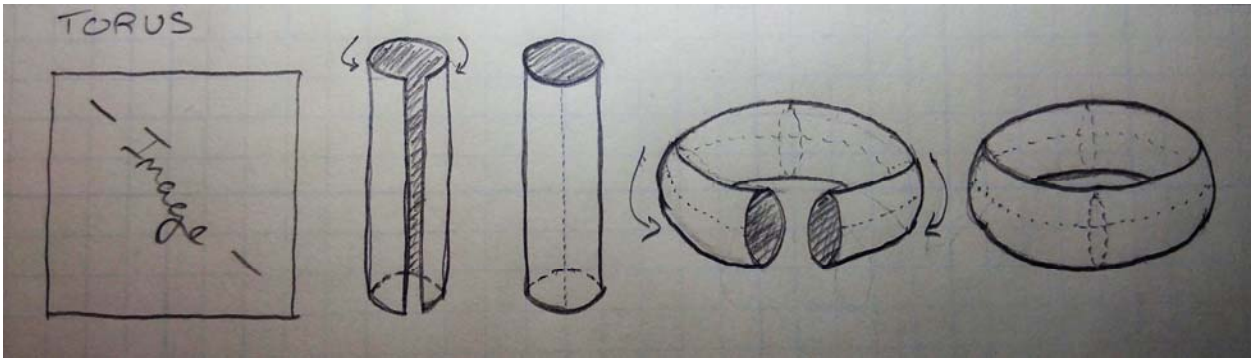

Figure 6: The process of wrapping an image onto a cylinder and then into a torus.

Given the equations designed above, we can then run a similar algorithm (see Appendix A) to use each point from the image to generate a point on the solid, then form triangles between these points to generate the final STL file. In some cases (e.g. cube, cylinder), a few extra points are added to close openings in the surface.

Results

The solids with images mapped onto them for each function designed are presented below. The images were generated by opening the solids' STL files in 3D Builder, a free 3D rendering program (often used for 3D printing). The physical 3D objects were printed on a Makerbot printer.

Cylinder

An example of an image mapped onto a cylinder is included below. The Seahawks logo was used as the image in this case.

Figure 7: The mapped cylinder model (Seattle Seahawks logo).

Overall, the cylinder mapping results in a clean model that shows nearly no distortion – the mapping only introduces a constant scaling in both the *x* and *y* dimensions.

Sphere

The mapped sphere below has a map of the world engraved into its surface.

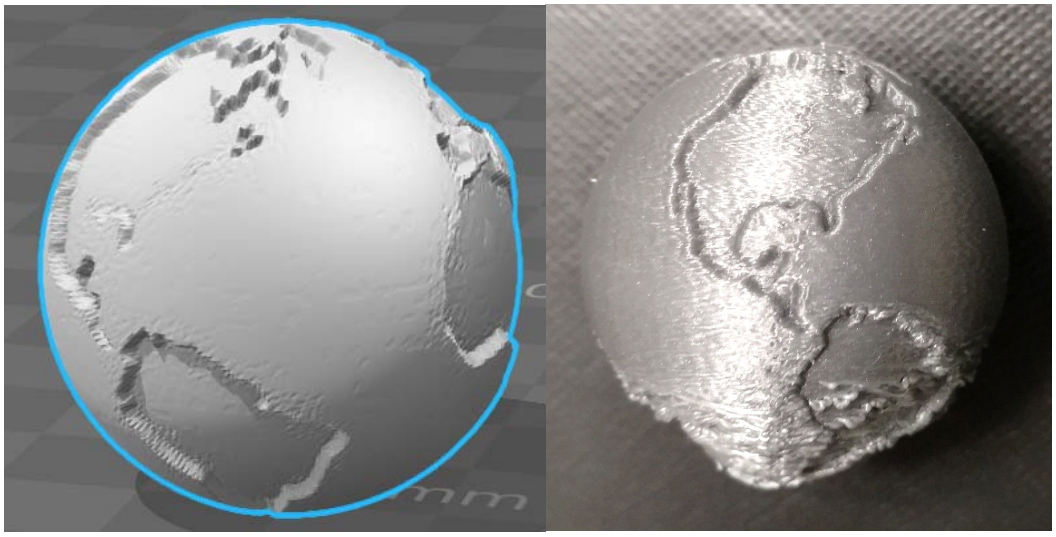

Figure 8: The mapped sphere model (world map).

The sphere mapping has a significant drawback in that it results in large distortion near each pole. While at any fixed *y*-dimension value there is no distortion along the *x*-dimension, the map becomes compressed to nearly a single point as it approaches each pole. Thus, the images used

should avoid any fine details toward the top and bottom of the image to prevent the loss of any important data.

Cube

The image included below was used to map four related images onto each face of a cube.

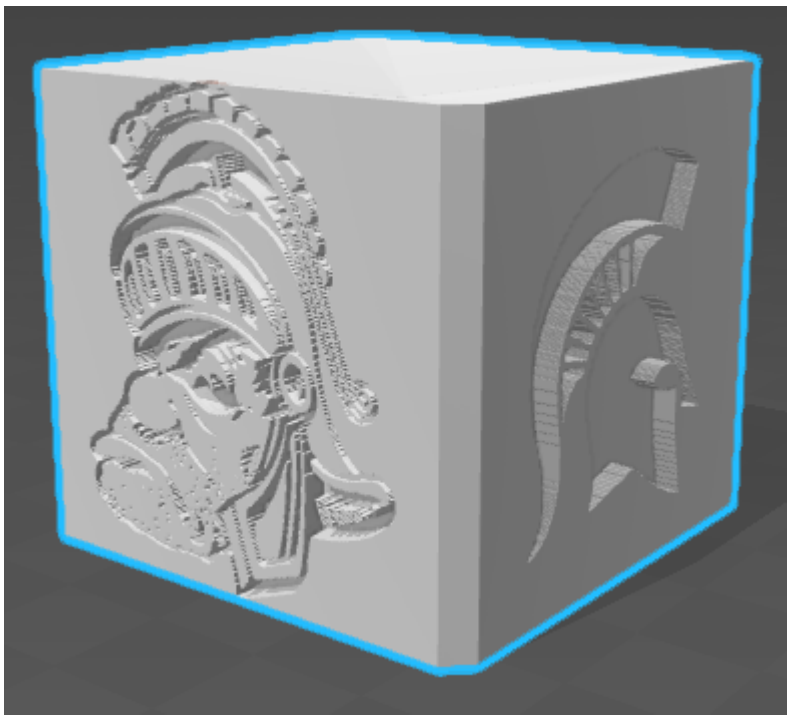

Figure 8: The mapped cube model (Michigan State Spartans logo).

The resulting cube has no distortion on the images on each face (only a constant scaling factor In both dimensions) but does result in a discontinuity along the edge between each face. This could be an issue if a single image is used to wrap around multiple faces, but still yields relatively clean results.

Torus

A mathematically interesting image was chosen for mapping onto a torus: a modified version of a complete graph with seven vertices. By using this modified version, the image becomes planar (in that the edges of the graph do not intersect) when mapped onto a torus. Note that on the torus, the image's top and bottom edges become adjacent and that the left and right edges become adjacent.

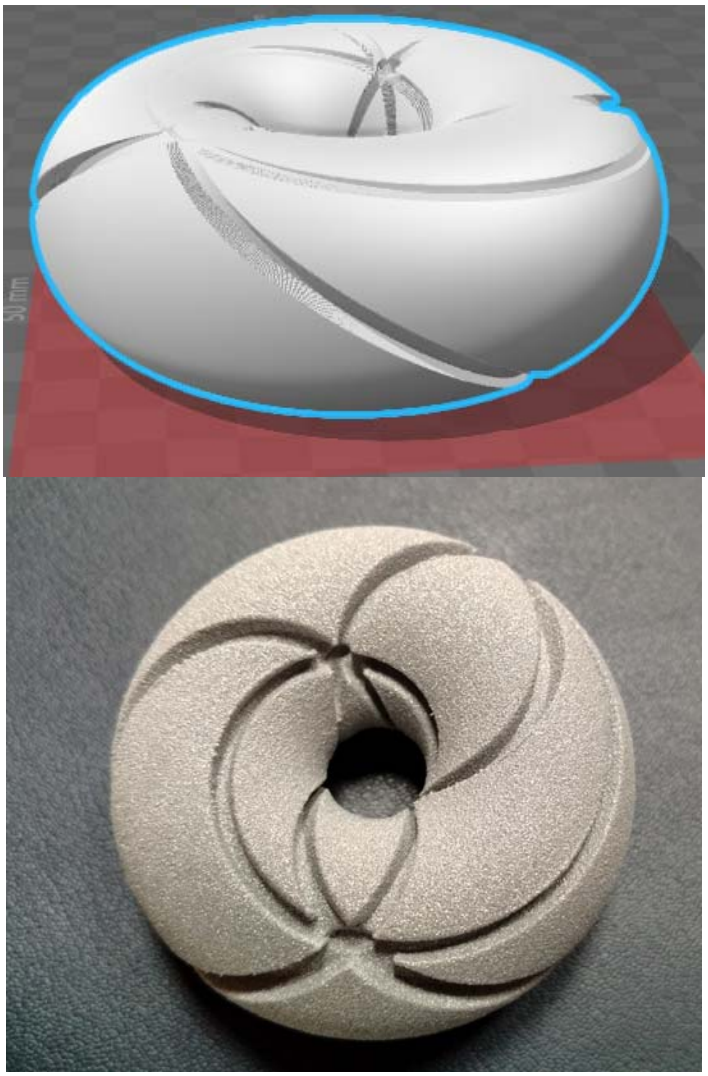

Figure 9: The mapped torus model (K_7 graph).

The torus mapping results in an interesting type of distortion. While at any fixed *y*-dimension in the image, the *x*-dimension is completely undistorted (there is only a constant scaling factor to reach the circumference at the corresponding ring in the torus). Along the y-dimension, however, there is noticeable distortion. While points on the "outside" of the torus are extruded outward such that the distance between points increases, points on the "inside" are extruded inward such that the distance decreases. Thus, the image becomes stretched on the outer surface and compressed on the inside.

Conclusion

The results generated by the algorithm explored in this paper is consistent with our initial intent, works for any rectangular grayscale image, and yields low distortion results (as long as the surface is low-curvature). Additionally, since the mapping is a direct result of a function designed with the end object in mind, there are no unexpected “glitches” in the solid that are caused by unpredictable software behavior. This method, however, falls short in its lack of flexibility to deal with complex solids and its requirement for the user to design a function for each new solid to be generated. Thus, while the method yields good results, it lacks the plasticity desired of a generic algorithm.

Future Work

Having discussed in great detail the method of using a function to generate solids with an image mapped onto the surface, the next reasonable step would be to investigate other methods for such a mapping problem, such as the second method described in the introduction which pushes the work done in this paper to yield a more flexible method that could be applied to any solid in general (not just those defined by clean functions). This method will tend to have higher distortion than the one studied in this paper, since it depends on the results generated by the first method (thus starting with some base distortion) and the surfaces mapped to will tend to have high curvature. The resolution of the models generated by this method should be similar to that of the method employed in this paper.

Another potential direction to consider is improvements to be made to the method used in this paper. While the results are consistent are relatively low in distortion, the solids it generates tend to have uneven surfaces and jagged edges, which may be undesirable for certain applications. Investigating a better way to “smooth” the generated surfaces between the generated points could improve the appeal of using this algorithm to map images onto a solid.

Sources

Aboufadel, Edward. *3D Printing A Pendant with A Logo.* Web. 10 January 2017. <https://arxiv.org/abs/1507.03102>

Burns, Marshall. *STL Format*. Fabbers. Ennex Corporation. Web. 10 January 2017. <http://www.fabbers.com/tech/STL_Format>

Christensson, Per. *Grayscale Definition*. TechTerms. Sharpened Productions, 01 April 2011. Web. 10 January 2017. <https://techterms.com/definition/grayscale>.

Appendix A: *Mathematica* Source Code

Code for Mapping an Image onto a Cylinder

```
(* The function to map a point onto a cylinder with radius r and height h *)
mappointtocylinder[x_, y_, g_, nx_, ny_, h_, r_] := {
 Cos[2*Pi*x/nx] * (r + g),
 Sin[2*Pi*x/nx] * (r + g),
 h * (y - 1)/(ny - 1)
}

(* The function to map an image ("points" array) onto a cylinder with radius r and height h *)
mapplanetocylinder[points_, r_, h_] :=
 Module[{cylinderpts, triangleindices, idxw, idxh, width, height, bottomidx, topidx, ptindex,
   triangleindex, idx1, idx2, idx3, idx4},

  height = Length[points];
  width = Length[points[[1]]];
  cylinderpts = ConstantArray[0, height*width + 2];
  triangleindices = ConstantArray[0, height*width*2];
  ptindex = 1;
  triangleindex = 1;

  (* Maps all of the points onto a cylinder *)
  For[idxh = 1, idxh <= height, idxh++,
   For[idxw = 1, idxw <= width, idxw++,
    cylinderpts[[ptindex++]] =
      mappointtocylinder[idxw, idxh, points[[idxh]][[idxw]], width, height, h, r]]];

  (* Connects all of the points into triangles *)
  For[idxh = 1, idxh < height, idxh++,
   For[idxw = 1, idxw < width, idxw++, {
    idx1 = (idxh - 1)*width + idxw;
    idx2 = idxh*width + idxw;
    idx3 = idxh*width + idxw + 1;
    idx4 = (idxh - 1)*width + idxw + 1;

    triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
    triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
   }]];

  For[idxh = 1, idxh < height, idxh++, {
   idx1 = idxh*width;
   idx2 = (idxh + 1)*width;
```

```
  idx3 = idxh*width + 1;
  idx4 = (idxh - 1)*width + 1;

  triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
  triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
 }];

 (* Adds the top and bottom to the cylinder *)
 (* Note: to improve the print quality, consider indenting the top and bottom of the cylinder by
  replacing the two points below with '{0, 0, 0.05*h}' and '{0, 0, 0.95*h}'. This allows the cylinder to
  sit flatter on the surface, even if the print is somewhat uneven *)
 bottomidx = height*width + 1;
 topidx = bottomidx + 1;
 cylinderpts[[ptindex++]] = {0, 0, 0};
 cylinderpts[[ptindex++]] = {0, 0, h};

 (* Connects the triangles on top and bottom *)
 For[idxw = 1, idxw < width, idxw++,
  idx1 = idxw;
  idx2 = idxw + 1;
  idx3 = (height - 1)*width + idxw + 1;
  idx4 = (height - 1)*width + idxw;

  triangleindices[[triangleindex++]] = {idx1, idx2, bottomidx};
  triangleindices[[triangleindex++]] = {idx3, idx4, topidx}
 ];

 idx1 = width;
 idx2 = 1;
 idx3 = (height - 1)*width + 1;
 idx4 = height*width;

 triangleindices[[triangleindex++]] = {idx1, idx2, bottomidx};
 triangleindices[[triangleindex++]] = {idx3, idx4, topidx};

 (* Forms a mesh out of all of the points *)
 MeshRegion[cylinderpts, Polygon[triangleindices]]
];
```

Code for Mapping an Image onto a Sphere

```
(* The function to map a point onto a sphere of radius r *)
mappointtosphere[x_, y_, g_, nx_, ny_, r_] := {
 -Cos[2*Pi*x/nx] * Sin[Pi*(y + 1)/(ny + 2)] * (r + g),
 Sin[2*Pi*x/nx] * Sin[Pi*(y + 1)/(ny + 2)] * (r + g),
 Cos[Pi*(y + 1)/(ny + 2)] * (r + g)
}

(* The function to map an image ("points" array) onto a sphere of radius r*)
mapplanetosphere[points_, r_] :=
 Module[{spherepts, triangleindices, idxw, idxh, width, height, bottomidx, topidx, ptindex,
   triangleindex, idx1, idx2, idx3, idx4},

  height = Length[points];
  width = Length[points[[1]]];
  spherepts = ConstantArray[0, height*width + 2];
  triangleindices = ConstantArray[0, height*width*2];
  ptindex = 1;
  triangleindex = 1;

  (* Maps all of the points onto a sphere *)
  For[idxh = 1, idxh <= height, idxh++,
   For[idxw = 1, idxw <= width, idxw++,
    spherepts[[ptindex++]] = mappointtosphere[idxw, idxh, points[[idxh]][[idxw]], width, height, r]
   ]];

  (* Connects all of the points into triangles *)
  For[idxh = 1, idxh < height, idxh++,
   For[idxw = 1, idxw < width, idxw++, {
    idx1 = (idxh - 1)*width + idxw;
    idx2 = idxh*width + idxw;
    idx3 = idxh*width + idxw + 1;
    idx4 = (idxh - 1)*width + idxw + 1;

    triangleindices[[triangleindex++]] = {idx1, idx3, idx2};
    triangleindices[[triangleindex++]] = {idx3, idx1, idx4};
   }]];

  For[idxh = 1, idxh < height, idxh++, {
   idx1 = idxh*width;
   idx2 = (idxh + 1)*width;
   idx3 = idxh*width + 1;
   idx4 = (idxh - 1)*width + 1;
```

```
  triangleindices[[triangleindex++]] = {idx1, idx3, idx2};
  triangleindices[[triangleindex++]] = {idx3, idx1, idx4};
 }];


 (* Closes the top and bottom to the sphere *)
 bottomidx = height*width + 1;
 topidx = bottomidx + 1;
 spherepts[[ptindex++]] = {0, 0, r};
 spherepts[[ptindex++]] = {0, 0, -r};

 (* Connects triangles on top and bottom *)
 For[idxw = 1, idxw < width, idxw++,
  idx1 = idxw;
  idx2 = idxw + 1;
  idx3 = (height - 1)*width + idxw + 1;
  idx4 = (height - 1)*width + idxw;

  triangleindices[[triangleindex++]] = {idx1, bottomidx, idx2};
  triangleindices[[triangleindex++]] = {idx3, topidx, idx4}
 ];

 idx1 = width;
 idx2 = 1;
 idx3 = (height - 1)*width + 1;
 idx4 = height*width;

 triangleindices[[triangleindex++]] = {idx1, bottomidx, idx2};
 triangleindices[[triangleindex++]] = {idx3, topidx, idx4};

 (* Forms a mesh out of all of the points *)
 MeshRegion[spherepts, Polygon[triangleindices]]
];
```

Code for Mapping an Image onto a Cube

```
(* The function to map a point onto a cube with side length s - Uses a piecewise function to
determine what side the point is mapped to *)
mappointtocube[x_, y_, g_, nx_, ny_, s_] :=
 If[x < nx/4, {-4*s*(x/nx), -g, s*y/ny },
  If[x < nx/2, {-(s + g), 4*s*(x/nx - 0.25), s*y/ny},
   If[x < 3*nx/4, {-4*s*(0.75 - x/nx), s + g, s*y/ny},
    {g, 4*s*(1 - x/nx), s*y/ny}]]]

(* The function to map an image ("points" array) onto a cube of side length s *)
mapplanetocube[points_, s_] :=
 Module[{cubepts, triangleindices, idxw, idxh, width, height, bottomidx, topidx, ptindex,
    triangleindex, idx1, idx2, idx3, idx4},

  height = Length[points];
  width = Length[points[[1]]];
  cubepts = ConstantArray[0, height*width + 2];
  triangleindices = ConstantArray[0, height*width*2];
  ptindex = 1;
  triangleindex = 1;

  (* Maps all of the points onto a cube *)
  For[idxh = 1, idxh <= height, idxh++,
   For[idxw = 1, idxw <= width, idxw++,
    cubepts[[ptindex++]] = mappointtocube[idxw, idxh, points[[idxh]][[idxw]], width, height, s]
  ]];

  (* Connects all of the points into triangles *)
  For[idxh = 1, idxh < height, idxh++,
   For[idxw = 1, idxw < width, idxw++, {
    idx1 = (idxh - 1)*width + idxw;
    idx2 = idxh*width + idxw;
    idx3 = idxh*width + idxw + 1;
    idx4 = (idxh - 1)*width + idxw + 1;

    triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
    triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
   }]];

  For[idxh = 1, idxh < height, idxh++, {
   idx1 = idxh*width;
   idx2 = (idxh + 1)*width;
   idx3 = idxh*width + 1;
```

```
    idx4 = (idxh - 1)*width + 1;

    triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
    triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
   }];

   (* Adds the top and bottom to the cube *)
   (* Note: to improve the print quality,
   consider indenting the top and bottom of the cube by replacing the \
two points below with '{-s/2, s/2, 0.05*s}' and '{-s/2, s/2, 0.95*
   s}'. This allows the cylinder to sit flatter on the surface,
   even if the print is somewhat uneven *)
   bottomidx = height*width + 1;
   topidx = bottomidx + 1;
   cubepts[[ptindex++]] = {-s/2, s/2, 0};
   cubepts[[ptindex++]] = {-s/2, s/2, s};

   (* Connects triangles on the top and bottom *)
   For[idxw = 1, idxw < width, idxw++,
    idx1 = idxw;
    idx2 = idxw + 1;
    idx3 = (height - 1)*width + idxw + 1;
    idx4 = (height - 1)*width + idxw;

    triangleindices[[triangleindex++]] = {idx1, idx2, bottomidx};
    triangleindices[[triangleindex++]] = {idx3, idx4, topidx}
   ];

   idx1 = width;
   idx2 = 1;
   idx3 = (height - 1)*width + 1;
   idx4 = height*width;

   triangleindices[[triangleindex++]] = {idx1, idx2, bottomidx};
   triangleindices[[triangleindex++]] = {idx3, idx4, topidx};

   (* Forms a mesh out of all of the points *)
   MeshRegion[cubepts, Polygon[triangleindices]]
 ];
```

Code for Mapping an Image onto a Torus

```
 (* The function to map a point onto a torus of large radius R and small radius r *)
mappointtotorus[x_, y_, g_, nx_, ny_, R_, r_] := {
 Cos[2*Pi*x/nx]*(R + (r + g)*Cos[2*Pi*y/ny]),
 Sin[2*Pi*x/nx]*(R + (r + g)*Cos[2*Pi*y/ny]),
 (r + g)*Sin[2*Pi*y/ny]
}

(* The function to map an image ("points" array) onto a torus of large radius R and small radius r *)
mapplanetotorus[points_, R_, r_] :=
 Module[{toruspts, triangleindices, idxw, idxh, width, height, bottomidx, topidx, ptindex,
       triangleindex, idx1, idx2, idx3, idx4},

  height = Length[points];
  width = Length[points[[1]]];
  toruspts = ConstantArray[0, height*width];
  triangleindices = ConstantArray[0, height*width*2];
  ptindex = 1;
  triangleindex = 1;

  (* Maps all of the points onto a torus *)
  For[idxh = 1, idxh <= height, idxh++,
   For[idxw = 1, idxw <= width, idxw++,
    toruspts[[ptindex++]] =
       mappointtotorus[idxw, idxh, points[[idxh]][[idxw]], width, height, R, r]]];

  (* Connects all of the points into triangles *)
  For[idxh = 1, idxh < height, idxh++, {
   For[idxw = 1, idxw < width, idxw++, {
    idx1 = (idxh - 1)*width + idxw;
    idx2 = idxh*width + idxw;
    idx3 = idxh*width + idxw + 1;
    idx4 = (idxh - 1)*width + idxw + 1;

    triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
    triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
   }];
   idx1 = idxh*width;
   idx2 = (idxh + 1)*width;
   idx3 = idxh*width + 1;
   idx4 = (idxh - 1)*width + 1;

   triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
```

```
   triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
  }];

  (* Connects the triangles between the first and last row of points *)
  For[idxw = 1, idxw < width, idxw++, {
   idx1 = (height - 1)*width + idxw;
   idx2 = idxw;
   idx3 = idxw + 1;
   idx4 = (height - 1)*width + idxw + 1;

   triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
   triangleindices[[triangleindex++]] = {idx3, idx4, idx1};
  }];

  idx1 = height*width;
  idx2 = width;
  idx3 = 1;
  idx4 = width*(height - 1) + 1;

  triangleindices[[triangleindex++]] = {idx1, idx2, idx3};
  triangleindices[[triangleindex++]] = {idx3, idx4, idx1};

  (* Forms a mesh out of all of the points *)
  MeshRegion[toruspts, Polygon[triangleindices]]
];
```

Code for Loading an Image into Mathematica

```
bound[x_] := x*255;
DataFront = Import["FilePath/image_file.jpg"];
SizeFront = Import["FilePath/image_file.jpg", "ImageSize"];
GrayFront = ColorConvert[Image[DataFront, "Real"], "Grayscale"]
ImagePts = Table[bound[ImageData[GrayFront][[i, j]]], {i, 1, SizeFront[[2]], 1}, {j,  SizeFront[[1]], 1, -1}];
```

Code for Exporting a 3DMesh as an STL File

```
(* Use one of the functions above that returns a MeshRegion in place of ‘mapplanetosolid’ here*)
Output = mapplanetosolid[ image, parameters ];
Export[ "FilePath\output_file.stl", Output, {"STL", "BinaryFormat" -> True} ];
```