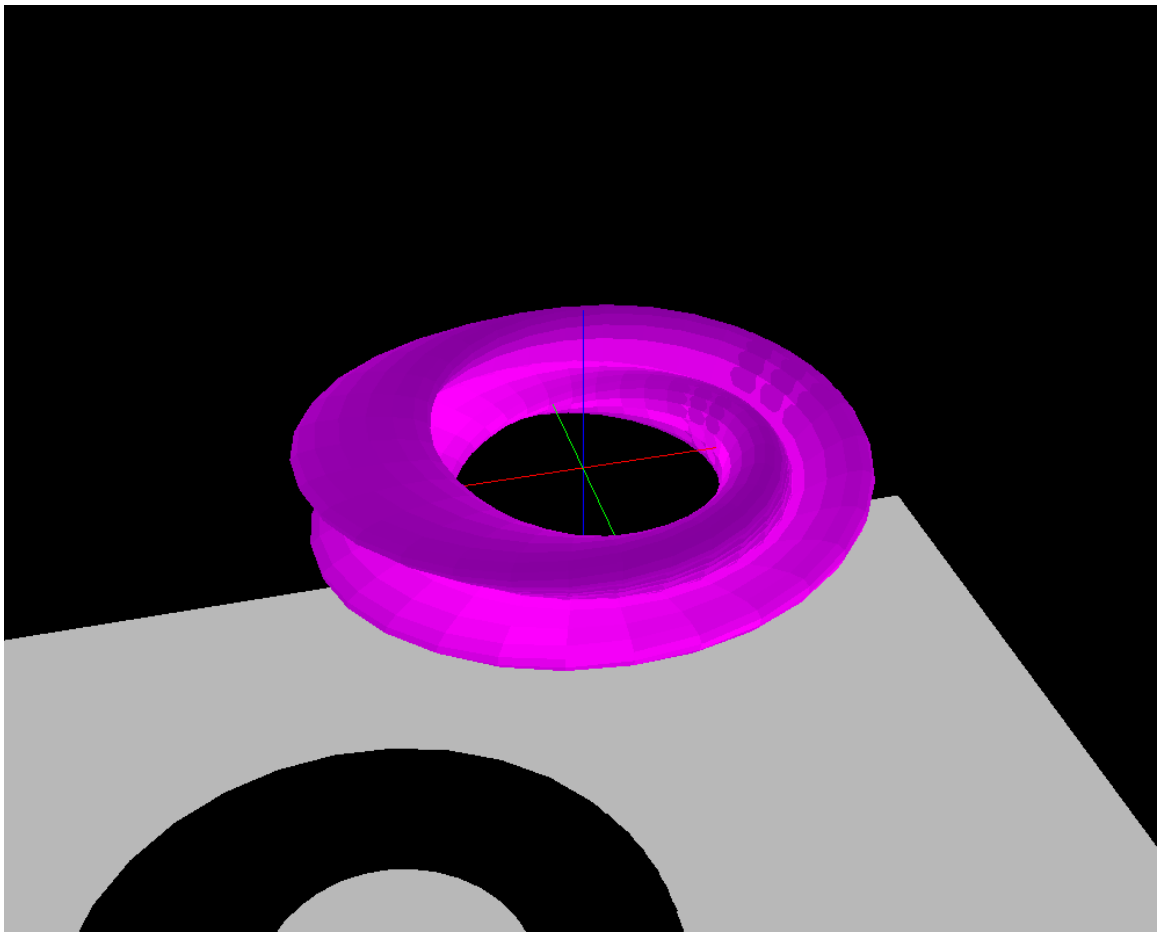
Shadows of 3D Surfaces

Molly K. Fane

December 16, 2015



Contents

1 Abstract

As an introduction into OpenGL and as an exploration into the creation of planar projection shadows, also called drop shadows, a semester-long project was undertaken with the following goal: Create a program that displays a point of light, a plane, a user-defined function, and the resulting shadow.This document outlines the process of competing project, nicknamed "shadow," how the program works, and the mathematics behind drop shadows.

2 Final Program Details

2.1 Program Functionality

The final program, ShadowPara.py, uses Python 2.7 along with PyOpenGL with GLUT in order to display the shadow scene. To define the displayed parametric function, users have access to the library of functions from the python math module as well as the built-in python functions.

The program requests three functions in terms of u and v set equal to x, y, and z to create a parametric surface. Although there are preset values, the user may change the bounds of u and v as well as the sampling rate of the functions in u and v used to create the approximate graphical representation of the parametric functions displayed. Finally, if one chooses, they may set the position of the point of light as well as the equation of the plane, however these have very functional preset values as well.

The display includes a set of x, y, and z axes colored red, green, and blue respectively. There are keyboard controls of the perspective of the display. The A, D, W, S and Q and E keys control translation and the J, L, I, K, and U and O keys control rotation. Each translates along or rotates about one of the three major axes.

2.2 Operation and Avoiding Errors

The functions must be entered as strings. I cannot stress this enough. The built-in eval() python function is used to assess the values of the x, y, and z at each pair of u and v values. In Python 2.7, the eval() function can only accept string input values, I cannot find a way around this issue even converting the input function into a string before using eval() does not work.[1] Therefore it is crucial that users enter their functions as strings, using single or double quotation marks around the input function. This problem is solved using Python3, in which the eval() function will accept a raw input that is not in string form.

Examples of interesting parametric surfaces and their bounds are given. However, one may enter any function they may choose as long it only references mathematical functions from the math module library or the python function library. Taking fractional exponents of negative numbers or dividing by zero will result in an error.

It is better to overdraw a function's bounds. Due to the approximate nature of the graph, off by one errors may arise if the range of u is not divisible by deltaU (the change in u between sample points). It is also best to ensure that the function is absolutely between the plan and the point of light, in order to ensure a logical shadow with no errors. No vector

from the Light point to a point on the function may be parallel to a point on the plane, this will result in an obstruction infinite quadrilateral in the shadow, obstructing results. Finally, be sure to use as few as possible sampling points. This means, choose a deltaU and a deltaV sufficiently large so that the computer can render the scne without crashing. The speed at which one can rotate or translate the perspective around the scene is directly related to the number of quadrilateral rendered in the scene. For fast navigation, use few sample points, i.e. large deltaVs and large deltaUs. The result varies depending on the specifications of the machine used to run the program.

3 The Mathematics behind Shadows

3.1 Types of Light

There are two different types of light: directional light and point light. Point light has light rays that emanate from a point in space. A lightbulb in a room is a good example of a point light source. Directional light is light that emanates from a point at infinity. The light rays that are produced are parallel.[2] A good example of directional light are the light rays from the Sun onto the Earth, because the Sun is far enough away from the Earth that the rays are approximately parallel when they hit the Earth.

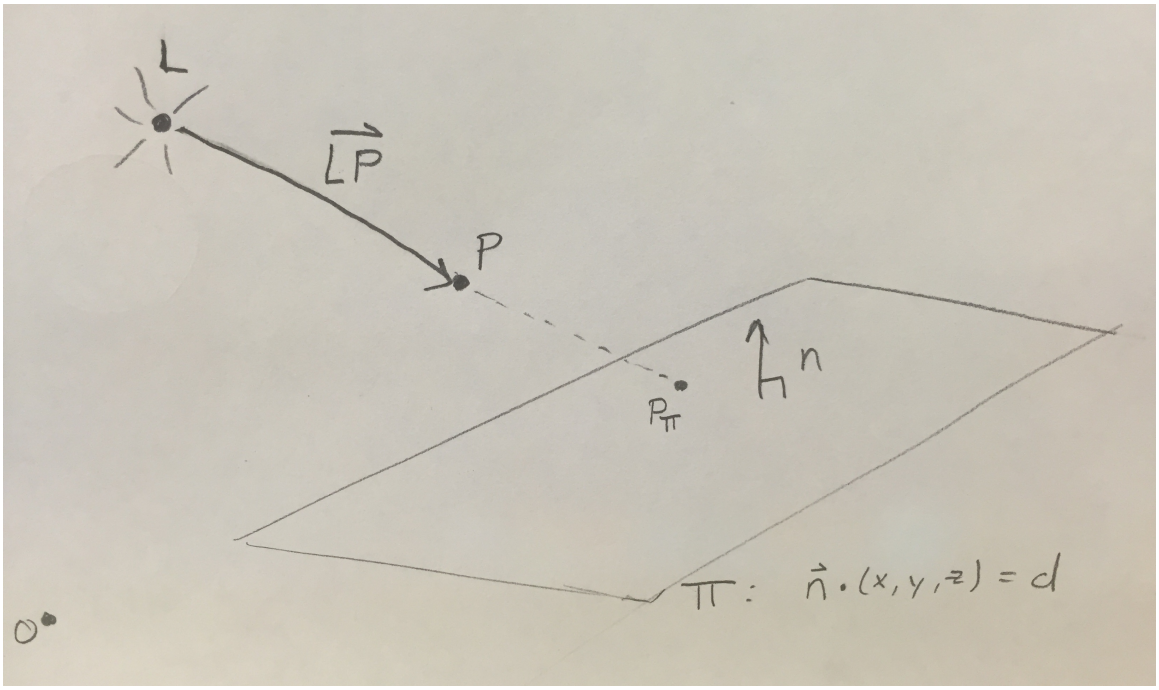
Points at infinity come from a field of geometry called "Projective Geometry." Unlike in the Euclidean Plane, the projective plane has the following duality:

Not only do two different points create a unique line, two different lines create a unique point (at their intersection).

Parallel lines intersect in projective space. They intersect at points at infinity. Similarly parallel planes intersect an lines at infinity in a 3D analogue to the projective plane. It can be said that projective space has more points that Euclidean space because of these points at infinity.[3]

3.2 Planar Projection

For the sake of interest in computer graphics: the discussion will be limited to only point light sources and 3D images made of sufficiently small polygons. The algorithm I used to create the drop shadows in my program is called the planar projection of shadows. The main concept of planar projection is to take each point of the polygons of the shadow-producing object and project it onto the plane from the light source.



In order to find the coordinates of P_π:
Find the equation of the line between the point of light and the object:

$$\vec{LP} : (x, y, z) = (\vec{P} - \vec{L})t + \vec{L}$$

Find the equation of the plane, and format it as such:

$$\pi : \vec{n} \cdot (x, y, z) = d$$

Substitute:

$$((\vec{P} - \vec{L})t + \vec{L}) \cdot \vec{n} = d$$

Solve for t:

$$t = \frac{d - \vec{n} \cdot \vec{L}}{\vec{n} \cdot (\vec{P} - \vec{L})}$$

Subsitute this t back into the equation of the line:

$$P_\pi : (x, y, z) = (\vec{P} - \vec{L}) \left(\frac{d - \vec{n} \cdot \vec{L}}{\vec{n} \cdot (\vec{P} - \vec{L})} \right) + \vec{L}$$

It is important to note that there is a divide by zero error if $\vec{n}$ is perpendicular to $(\vec{P} - \vec{L})$, as there should be, because this situtaiton would create an infinite shadow.

In my program, I use this algorithm on each sampled vertex in order to find its shadow on the plane below. The quadrilaterals drawn in the function above, are redrawn between vertexes with the same indices as in the function above, but at their shadow points on the plane below creating the solid 2D shape of the shadow.

4 How the Code works

4.1 Parser

At the core of every program that graphs functions is a parser. I use the python eval() function as a parser. The default eval() function takes a string, reads the recognizes the functions called, the variables named, and integers and floating point decimal numbers named. It can then execute the commands in order to return the proper value as an output. In my program, I take advantage of the eval() function to essentially evaluate a function at many different values of u and v.

4.2 Step by Step Explanation

The follwoing is a step by step explanation of the core functions of my program. The user inputs a function for x, y, and z each in terms of u and v as well as the range of values for u and v to graph. A parsing algorithm samples the function at points deltaU and deltaV intervals apart. These samples are added to a list called vertices as the list [x,y,z,a,b] where a and b are index variable that count the number of points sampled past the initial point in the u and v directions respectively.

Now this list of vertices must become quadrilaterals. Each quadrilateral is made up of four points indexed adjacent to each other. If (a, b) are the values of the indices of the bottom left vertex of a quadrilateral, than the top left is $(a, b+1)$, top right is $(a+1, b+1)$, and the bottom right vertex is $(a+1, b)$. A relatively inefficient algorithm (it could be made more efficient if it were affecting runtimes) finds the vertices with the correct a and b indices to form the quadrilateral and those four vertices are sent through glBegin(GL_QUADS) which takes four vertices at a time and draw the quadrilateral between them.[4] During glBegin() sequence, each quad is colored according to z value of the bottom left vertex. The glBegin() sequence is in the drawQuads() function which is called in the glutDisplayFunc() callback function.

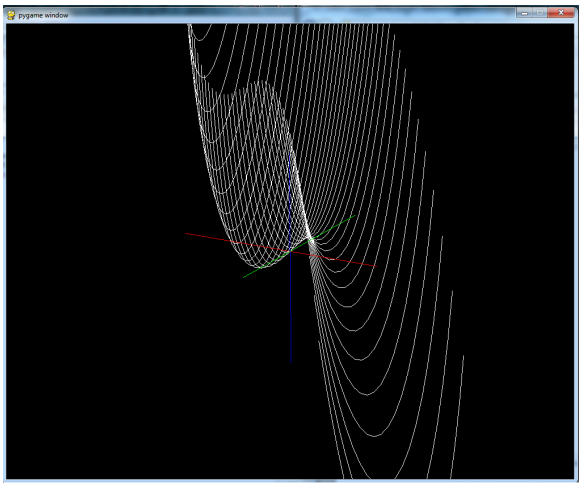
From the vertices list, the shadow vertices list is defined following the algorithm discussed in the mathematics section, careful to avoid a divide by zero error. In order to avoid depth error, since the plane and the shadow quads will lay in the same plane, the shadow is slightly raised, meaning t is reduced ever so slightly. A shadow vertex is defined for each real vertex and is placed into the shadowverts list the same way the vertices list was formed. And the shadowverts list is also sent through drawQuads() with a different shading so that shadow is dark.

These functions are, together, the core of my project. Together, they create the function and its shadow. (Though dont forget this is bundled along with the aid of all of the OpenGL intricacies that allow the function and its shadow to be displayed).

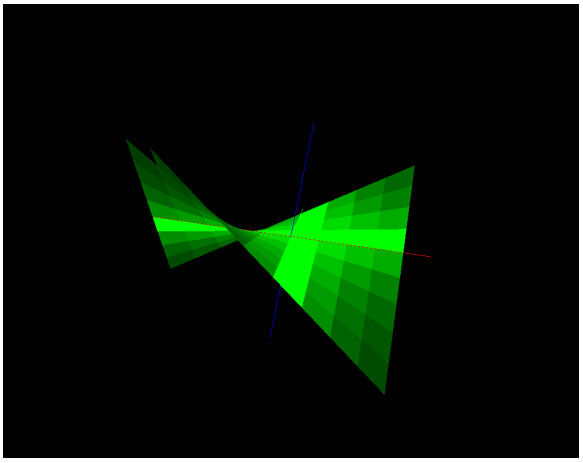
5 Process

The first form of the Shadow project was a 3D Surface grapher, called shadowPygame.py, that used Python 3.5 and a display module called pygame. The program requested a func-

tion of the form $z = f(x, y)$ and it returned a keyboard rotatable display of the graph of the functions surface made up of traces of the surface in the direction of planes parallel to the yz-plane. This program used OpenGL to create edges approximating each trace of the function as a line. This was a typical output of the program:



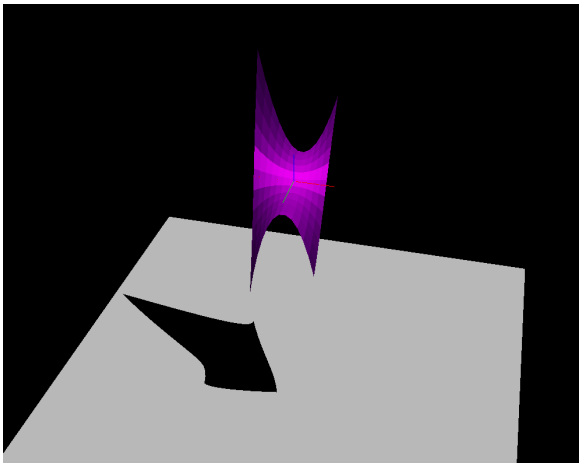
Soon, unsatisfied with simple line traces, and aspiring to create a program similar to DP-Graph, I altered the program to use quadrilaterals instead of edges, each quadrilateral requiring four correctly ordered sample points in order to define a part of the surface. This program was called shadowQUAD.py, returning images that look like this:



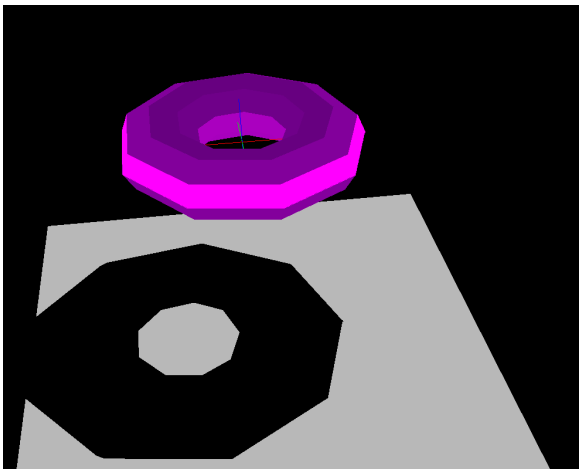
I decided it was important to use software already installed on the classroom lab macs so that I may run my programs in the lab as well as on my computer. This meant I had to convert my program to run using Python 2.7. I also had to do away with pygame and instead use GLUT to display my OpenGL content. This really forced me to learn about how to set up an OpenGL GLUT scene including what functions you must call in order to initialize a window and display content using GLUT. After a long and arduous conversion process, I had created a similar surface grapher able to run on lab macs called shadowQuadGLUT.py

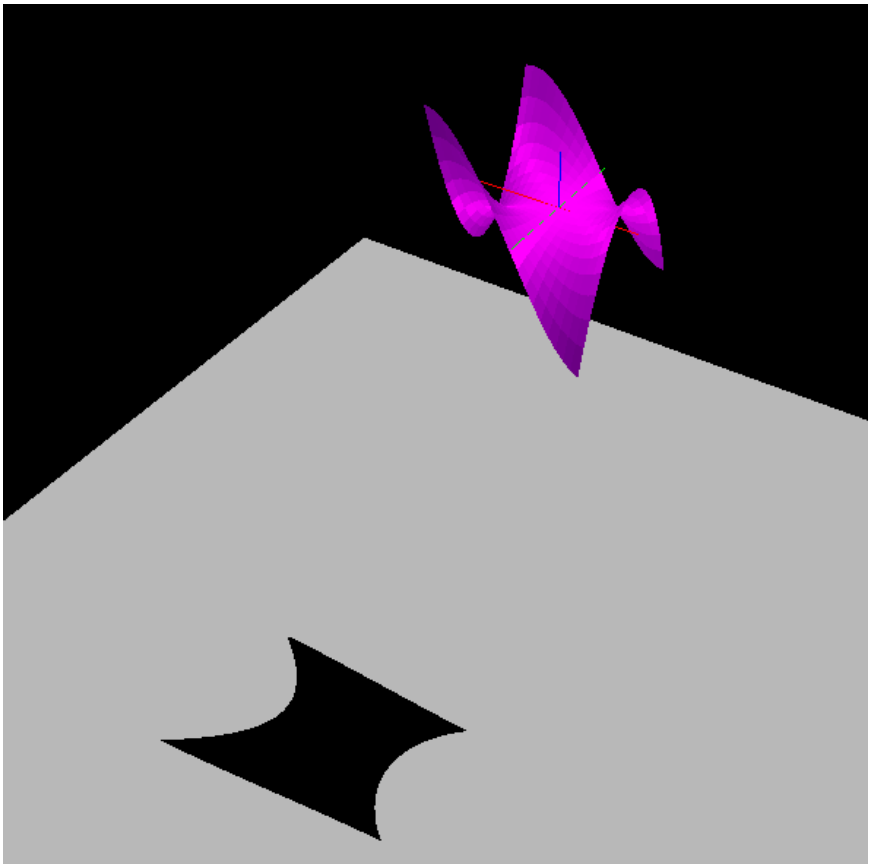
Finally I was ready to create some shadows. I created a quadrilateral to represent a

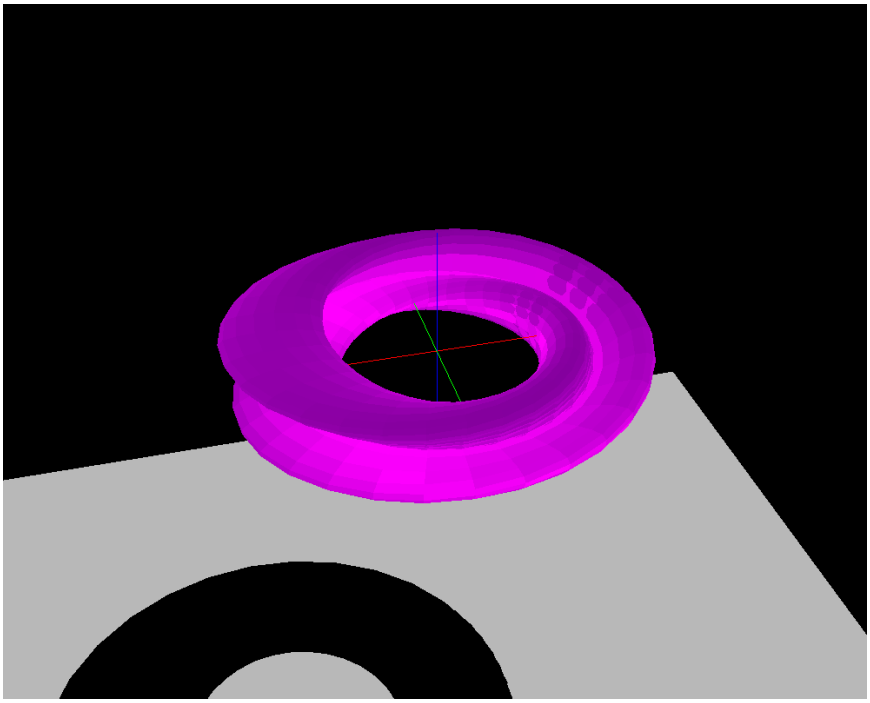
plane and a yellow asterisk to represent a point of light. Having previously worked out the algorithm with which one may create a drop shadow, I translated this algorithm to python code, translating each vertex on the surfaces to a vertex lying on the plane. This was the first program to achieve the core goal of the project. It is called shadow2.py in the repository. Here is a picture of what the first drop shadow program creates:



Shadow2 has one flaw. It will only accept functions in terms of only x and y and set equal to z. While I tried new functions, I discovered this to be increasingly limiting in what was possible to display. I decided that, in order to add versatility to the grapher portion of the project, I created a version of shadow2 with a parametric grapher. This was the final program created for this project. It is called shadowPara.py. The great thing about the parametric grapher is that it can graph parametric surfaces using u and v as well as functions set up as $z = f(x, y)$, simply by setting $x = u$, $y = v$ and $z = f(u, v)$. Here are some of the functions and theirs shadows I have graphed using shadowPara.py:







This is a Klein Bottle, one of the many suggested surfaces available in the intructions. [5]

6 Bibliography

1. Python.org.*Built-in Functions.* (n.d.). Retrieved December 16, 2015, from https://docs.python.org/2/

2. Ute Rosenbaum, A. (1998). The Axioms of Projective Geometry. *In Projective Geometry: From Foundations to Applications* (p. 8). Cambridge: Cambridge University Press.

3. Birn, Jeremy. (2006) *Digital lighting and rendering Berkeley*, CA : New Riders.

4. OpenGL.org.*GlBegin.* (n.d.). Retrieved December 16, 2015, from https://www.opengl.org/sdk/docs/

5. Karcher, Hermann.(2004).*Klein Bottle*,virtualmathmuseaum.org, from virtualmathmuseum.org/Surface/klein_bottle/klein_bottle.html.