Expanding on the Pythagorean Theorem

New Mexico

Supercomputing Challenge

Final Report

March 3, 2006

Team Number 75

Onate High School

Team Members:

Kyle Fitzpatrick

Brett Beckett

Meghan Scott

Kevin Christeson

Teacher:

Donald Downs

Josefina Dominguez

Table of Contents

Executive Summary

For centuries man has used the Pythagorean Theorem ($a^2 + b^2 = c^2$) to explain the relationship between the sides of any right triangle. The Pythagorean Theorem has many applications in real world problems, but the potential ability for this equation to define *any* triangle remains unexplored. By replacing the exponential constant (2) with a variable (x), we form $a^x + b^x = c^x$. Since x is in every term of the equation, solving for x analytically is impossibly difficult. Our group solved this problem using computationally intensive methods that provided an extremely accurate result.

The goal of this project was to create a precise model of the modified Pythagorean Theorem using a combination of Newton's method and a 3D representation with a variable level of detail. To find the exponent on a given triangle, we used Newton's method for approximating solutions of the equation. After finding a sufficient number of x values we used a unique graphing method that allowed us to observe the changes in the variable x relating to the sides of a triangle, for both two and three-dimensional graphs could not display enough variables. This graph will implement the RGB color scale, as well as the use of a three dimensional surface to demonstrate the relationship between the sides and their respective exponent. The RGB color scale assigns a unique shade to every individual triangle, and a height dimension represents the exponent x. To obtain an

accurate result, a computer was needed to perform the numerous calculations involved in solving Newton's method many times in a reasonable amount of time.

Introduction

This project was motivated by the curiosity of the group members. It will simply create a mathematical model of the relationship between an exponent and its respective sides. Remember, the most famous mathematical problem in the past several centuries was Fermat's last theorem. This problem answered nothing but the curiosity of the mathematicians who solved it and the many that could not before computers. We created this project as mathematicians, not as scientists, and have a sincere interest in the outcome of this project.

Begin with a simple 180-degree triangle (a line). If we divide the line (c) into two segments, a and b are formed. Therefore, $a^1 + b^1 = c^1$ is formed. Comparing this to the Pythagorean Theorem $a^2 + b^2 = c^2$ caused us to think about what could be in between the exponents 1 and 2. We then formulated our hypothesis $a^x + b^x = c^x$. Furthermore, we calculated that an equilateral triangle has no solution for x. Also, we proved an analytical solution for an isosceles triangle, which would aid us in solving our hypothesis.

After proving that exponents could exist in isosceles triangles, we elaborated on our hypothesis by stating that any triangle has a unique variable that can be graphed. To find accurate exponents, precise approximations using Newton's Method were discovered. Newton's method is an iterative tangent line approximation of a value, and can be very accurate if repeated many times.

To represent our results accurately, a two-dimensional graph could not be used, for we had more than two variables. A three-dimensional graph would not be sufficient either, since we had to represent each individual triangle and its respective exponent in an

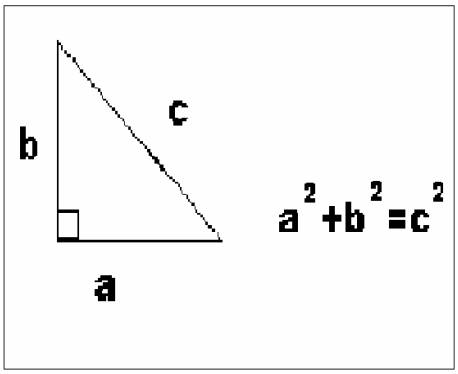
easy-to-read format. We created our own graphing system to represent our results. This graph uses the RGB color scale to represent each individual triangle by assigning a certain amount of color to each angle to obtain a shade unique to that triangle. Then, the triangle's respective exponent will be assigned as the height variable, allowing us to easily see any patterns that may exist in the graph. This custom graphing system will allow individuals to easily interpret the information by essentially reducing the results down to three variables.

Description

Most people know what the Pythagorean Theorem is. And most everybody has accidentally applied it to more than just right triangles. That little mistake started to make us think, why right triangles, why not *all* triangles? With much thought, and much trial and error, we started to realize what to do.

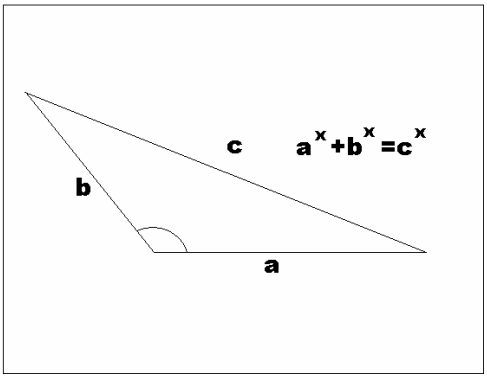
The Pythagorean Theorem states that in a right (90 degree) triangle, the sum of the two legs squared is equal to the third leg squared, like so in **fig-1.**

fig-1



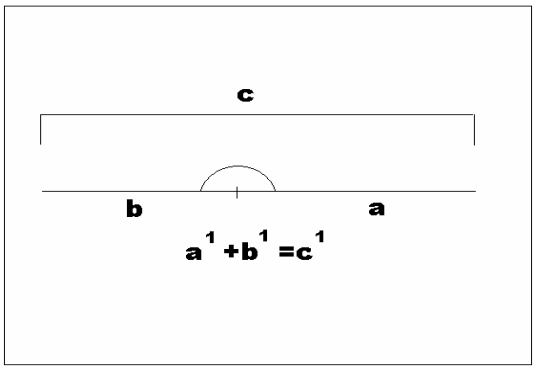
We wanted to know what would happen if instead of a 90-degree angle, you changed it to some other angle, and found a variable exponent to use like in **fig-2**

fig-2



We thought this because what happens if you expand the angle even more, to 180 degrees? Well, let's look at **fig-3** to see.

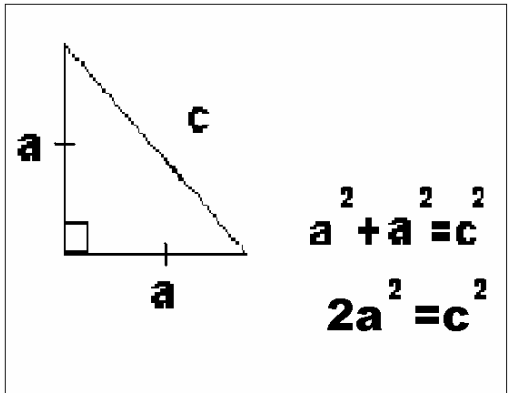
fig-3



You get a line. So at 90 degrees you use $a^2 + b^2 = c^2$ and at 180 degrees you use $a^1 + b^1 = c^1$**.** Do you see the similarity; 90 degrees and you use the exponent 2, 180 degrees and you use the exponent 1, is that coincidence or does every angle have an exponent? Our thesis is that $a^x + b^x = c^x$ in all angles except for equilateral triangles, two times any non-zero number cannot equal the same number.

We wanted to work with a limited triangle so there weren't any twists, so we decided to work with isosceles triangles (triangles with two sides the same.) That meant that we had to rearrange the formula's a little bit. Now look at **fig-4** to see the modified isosceles Pythagorean Theorem.

fig-4



Now it has changed from $a^2 + a^2 = c^2$ to $2a^2 = c^2$. This means that our thesis has changed a bit too.

Now it is $2a^x = c^x$**.** Now we want to know what the exponent equals, so let's solve for "x." Look at **fig-5-1**

fig-5-1

where theta is any angle ≠ 60

$$2a^x = c^x \qquad \ln(2) = x \ln\left(\frac{c}{a}\right)$$

$$2 = \frac{c^x}{a^x} \qquad x = \frac{\ln(2)}{\ln\left(\frac{c}{a}\right)}$$

$$2 = \left(\frac{c}{a}\right)^x$$

$$\ln(2) = \ln\left(\frac{c}{a}\right)^x \qquad \boxed{x = \frac{\ln(2)}{\ln\left(\frac{hyp}{leg}\right)}}$$

With further evaluation, we proved:

$$\text{Exponent} = \frac{\ln(2)}{\ln\left(\frac{\sin\theta}{\sin\left(\frac{180-\theta}{2}\right)}\right)}$$

This was a great step for us in searching for the answer to solve any triangle, but we still had a lot of thinking to do.

So given any set of numbers a, b, c, we need to use computers to approximate "x" in $a^x + b^x = c^x$

We simplified the problem by dividing both sides by c^x

$$\frac{a^x}{c^x}+\frac{b^x}{c^x}=\frac{c^x}{c^x}$$

Which can be simplified further by using a new a_1 and b_1.

$$a_1=\left(\frac{a}{c}\right) \qquad b_1=\left(\frac{b}{c}\right)$$

$$a_1^x+b_1^x=1$$

This equation is much simpler; we can now use Newton's Method to solve for the exponent. Newton's Method is an iterative tangent line approximation; you give a guess(n), and it narrows your guess down until it gets an accurate approximation.

Newton's Method is described by $x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$

We can rewrite our simpler function as $f(x)=a^x+b^x-1$

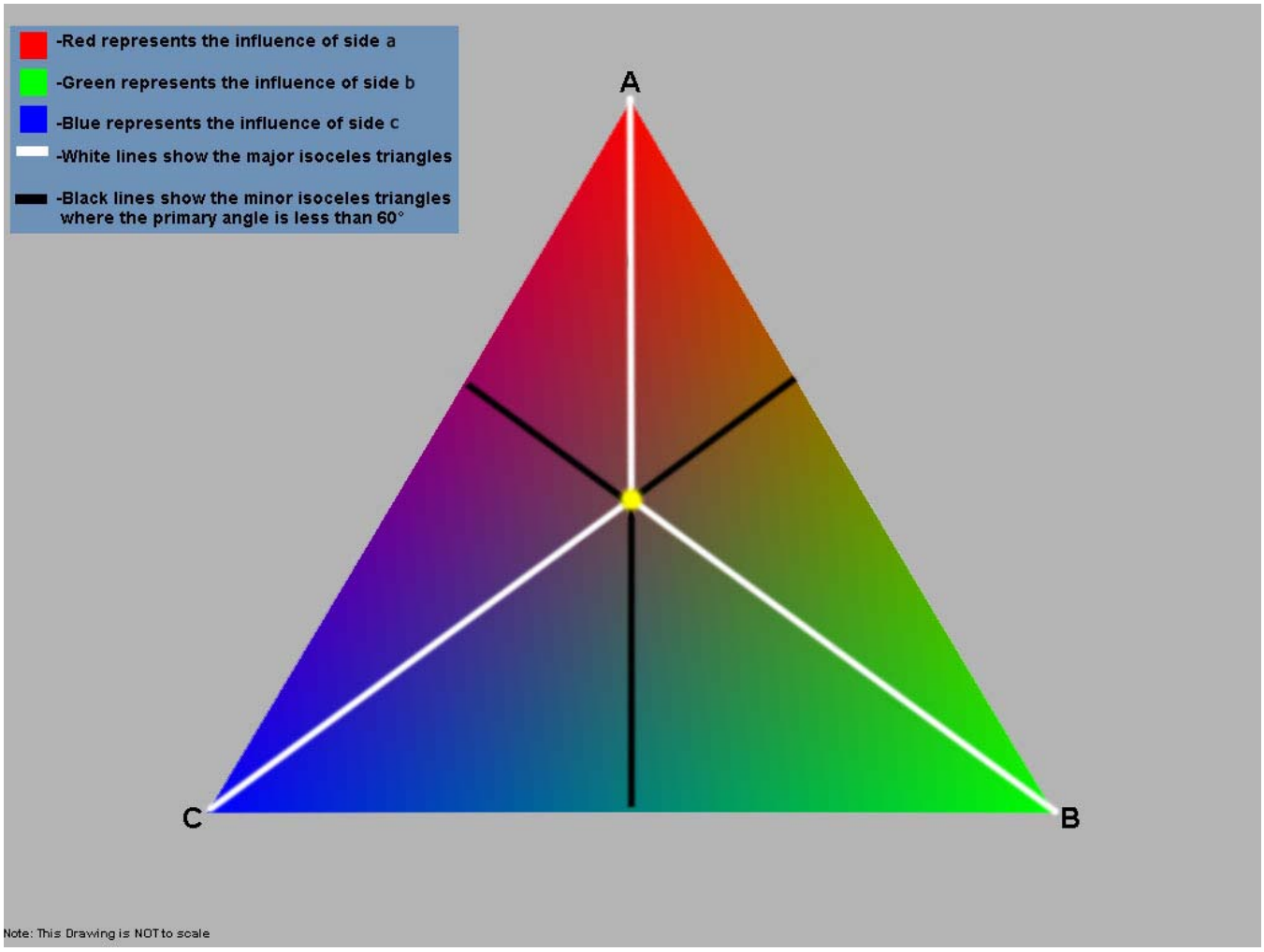
Then $f'(x)=a^x\ln a+b^x\ln b$

After substituting these functions in, we can now find a solution for a, b, and c.

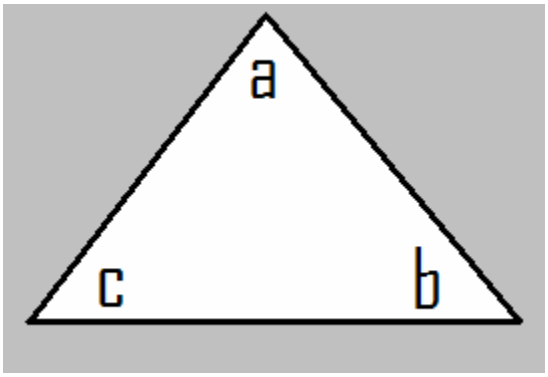
Displaying Results

Now that exponents can be found, we had to consider displaying our results. Since we couldn't solve it analytically, we weren't going to have a simple equation. We have to display with a graph. A two dimensional graph won't work, not even a three dimensional graph could work since there are 4 variables. We may have used a three dimensional graph plus time, but that would be very difficult to read. So we had to think of a new way to graph our results.

Graph



Any Triangle



We assigned the three angles in a triangle three different colors, red (angle A), green (angle B), and blue (angle C). A color represents a certain triangle, which is influenced by the triangles major angles. For example, if the largest angle on a triangle is “A”, then on our graph, that triangle would be more towards the red corner. If the largest angle was “A” and the second largest “Bs”, then the point would be an orange color on the upper right of our graph. With that understanding; the very middle, having equal amounts of red, green, and blue, would be an equilateral triangle. The white and black lines, having a variable amount of one color and the same amount of the other two colors, would have two angles of the same, being an isosceles triangle. So basically, every possible triangle has a certain color that can be found on our graph.

Now we can represent any triangle, but we still haven’t involved the exponent. We represented the exponent by using a height elevation on any point on our graph. The graph is colorful, and 3 dimensional. You find the triangle in the graph by it’s color, and however many units high that point is, that’s the exponent that you multiply the two known sides by to get that root of the third side.

Implementation

The next portion of our problem explains the methods of transposing the above explained concept into code. There are two parts of the program that had to be written: gathering results and displaying that data. Instead of separating these tasks into different programs, we decided to combine the entire process into one central effort for efficiency. The entire program, with the exception of the windows code (which has been removed because of its length and relatively unimportant monotony), is in the appendix section of this report.

The implementation of the color triangle has several parts which are all included within the main program. The foremost is the Point3d class, which holds the x, y, z coordinates, RGB color information, and the lengths of sides a and b of the triangle that point represents. In the initialization portion of the program an two-dimensional array of Point3d's is allocated which has [3][4^x] points, where x is the number of subdivisions for the central triangle. The next portion of the program is to take a given equilateral triangle and subdivide it x number of times using a recursive function called DivideMatrix(). DivideMatrix() calls itself numerous amounts of times, taking midpoints of the initial triangle and making smaller triangles in each iteration. The DivideMatrix() function writes to the point array in such a fashion that each row of the array contains one triangle. The next called function is BuildDefinition(), which sorts through each point in the Point3d array and finds the RGB color information, along with the solution to x of $a^x + b^x = c^x$ at that point. The color information and the solution to x (which is derived from the color information) is based off of the point's physical position on the subdivided equilateral triangle. Three equations are used to calculate the angles opposite to their

corresponding side a, b, or c. These equations can be found inside the GetAngle() function The angle information is then transformed into color information by the function GetColor(). After the angles have been calculated, a possible solution of x can be found. For this, the function GetRoot() is called. This function finds the x solution to $a^x + b^x = c^x$ if sides a, b, and c. GetRoot() sets side c equal to 1.0, and uses the Law of Sines to find sides a and b. The function then uses Newton's Method for approximating roots to find a suitable exponent (x) for sides a, b, and c, if possible. There are several instances though, where no real-number solution exists for x, as discussed in the results. After this process is ran, the Point3d array is ready to be transposed into OpenGL (which is the language we used to make a 3-D representation of our results). The last function that is called is BuildLists() which compiles the point information into an OpenGL object referenced later by the DrawGLScene() function. The DrawGLScene() function draws the array over and over in real-time, and also translates and rotates the view based off of user input.

Results

After designing, programming, and debugging the program, we found our results to be surprising, and much more exciting than we had earlier anticipated. As defined in the description portion of this paper, we are using a color-coded triangle where each color represents one of three angles on any given triangle, and the height value describes one solution to the modified function.

When graphed, the function of the exponent for any isosceles triangle is identical to the two dimensional cross section shown in figure 6. You will notice by looking at figure 6, that the exponent has both positive and negative results. When $a + b > c$,the exponent is negative because the number has to decrease. $a + b > c$ when the angle between a and b is less than 60 degrees. When the angle is greater than 60 degrees, the exponent is positive.

As seen in figure 9, there are two holes in this graph where the exponent does not exist: when the blue value (angle C) goes beneath the midpoint (equilateral triangle) it is no longer the dominant angle of the triangle, so the red (angle A) and green (angle B) angle values must compensate, which is not always a possibility. Red and green can only compensate when angle C is either the largest or smallest angle in the triangle.

Fig.6

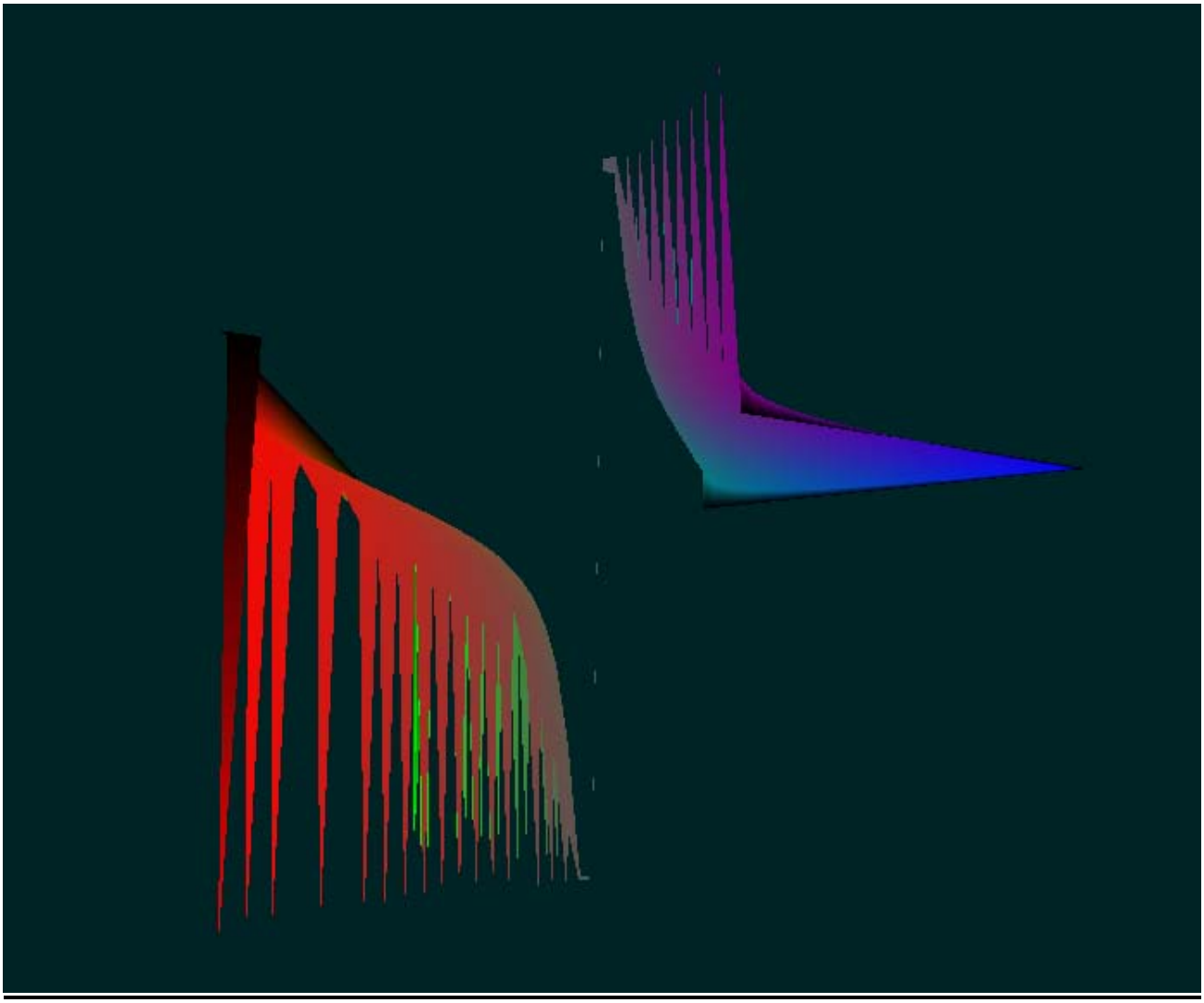


Fig.7

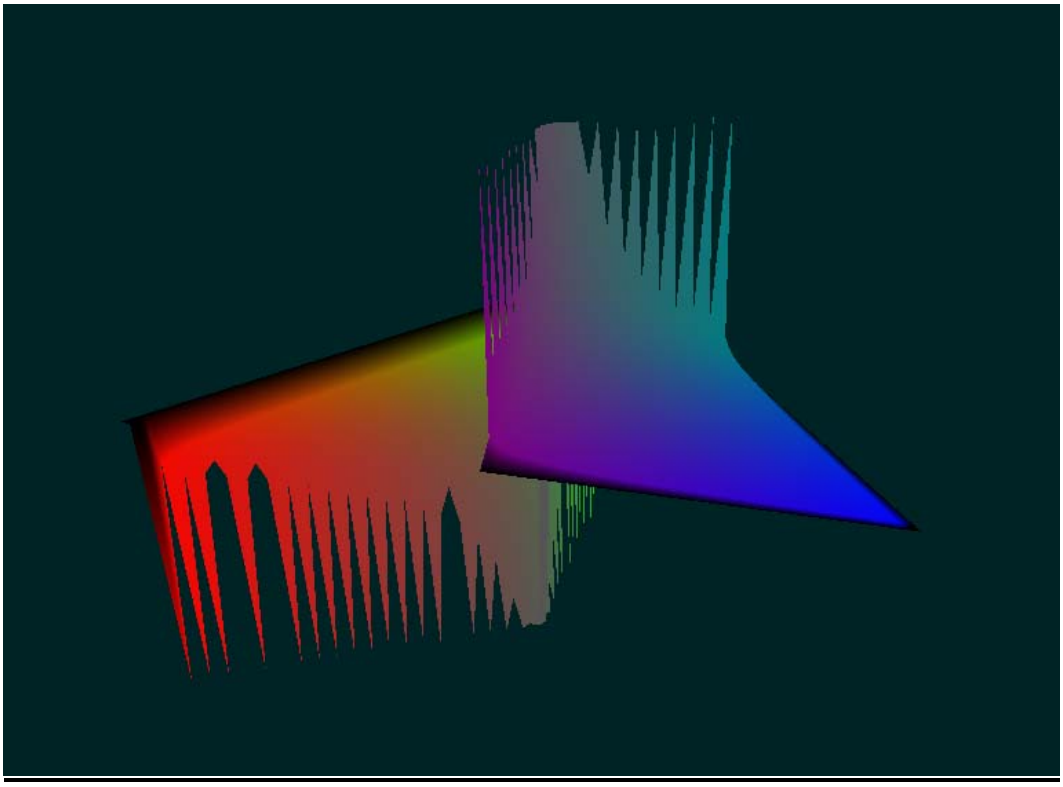


Fig.8

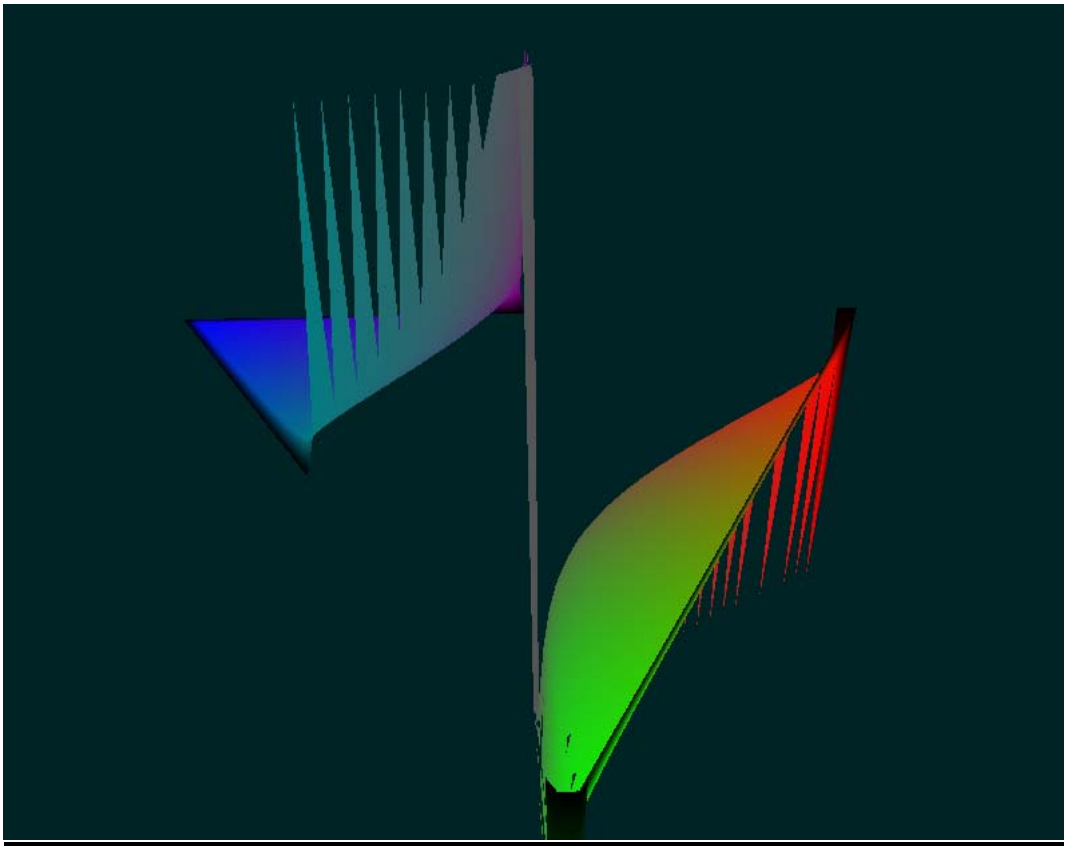
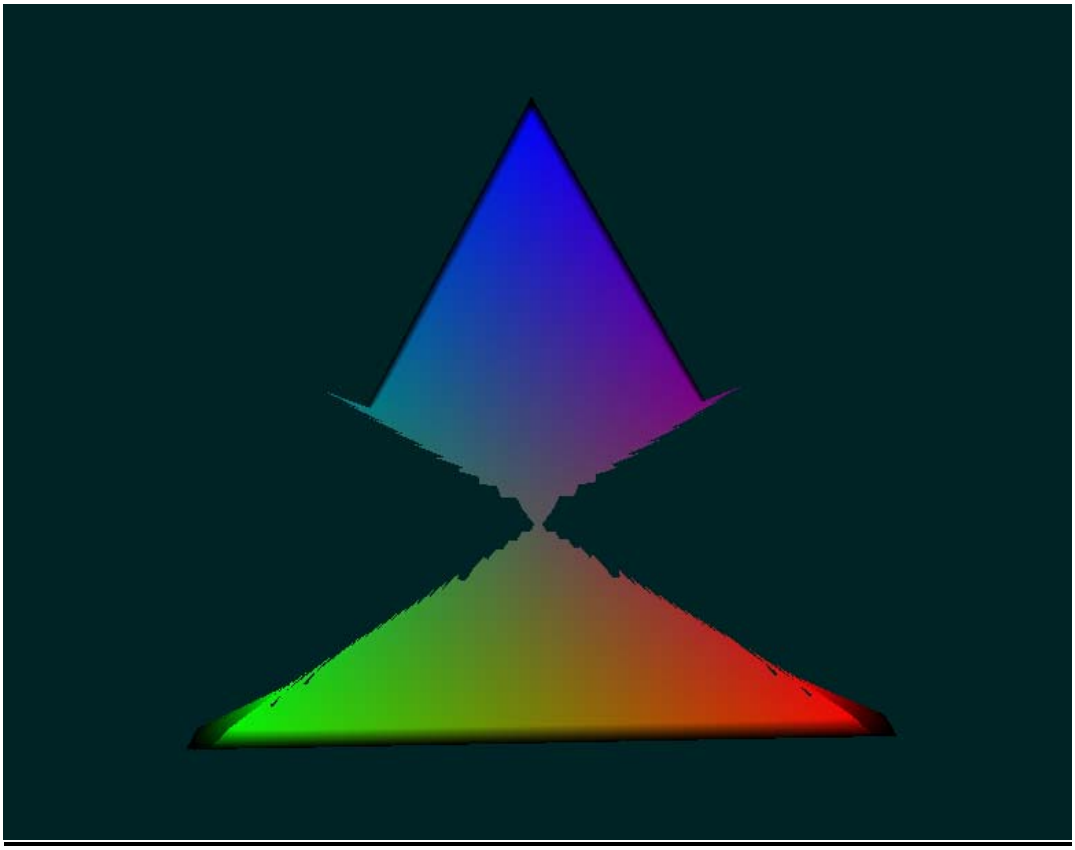


Fig.9



Conclusion

Some people have asked members of our group “Does this problem have any particular purpose in the real world, or is it simply theory and numbers?” to which we grin slightly and begin explaining the importance of any discovery, and how our problem shares commonality with one of the greatest mathematical puzzles in the last few centuries. Jules Henri Poincaré, a mathematician, theoretical physicist, and a philosopher once said, “The mathematician does not study pure mathematics because it is useful; he studies it because he delights in it and he delights in it because it is beautiful.” Like Poincaré, we all share a passion for discovering the unknown and admiring its allure, but more specifically the explanation of a new relationship involving triangles, the shape which has had an immeasurable impact on mathematics as a whole. Pythagoras' Theorem, which is one of the most commonly taught postulates because of its simplicity, serves to explain the relationship between the sides of right triangles. What our group has done is expand Pythagoras' Theorem to encompass any triangle, with the exception of an equilateral triangle. Granted, what we have found can be explained using other relationships, such as the Law of Sines and Cosines, but not in the same fashion. Currently, we have not found any practical purpose suited for the modified Pythagorean Theorem, but it serves as one more bit of knowledge advancing human understanding.

As earlier explained in the description section of our paper, we have implemented a non-standard method of graphing our function (which is a four-variable equation after canceling-out one variable with substitution) that is far more efficient and easier to visualize than a natural four-dimensional function.

Recommendations

Our problem found many solutions for $a^x + b^x = c^x$, and it opened up an endless field of possibilities. Such possibilities include complex solutions, using different methods to graph results, and most importantly the possibility of finding a natural function to $a^x + b^x = c^x$.

Acknowledgements

We would like to greatly thank our math teacher Mr. Downs for helping us get through our tough spots. We want to thank Mr. Dominguez, our parents, and all of the people involved in the Supercomputing Challenge.

Sources

Kempf, Frazier; OpenGL Reference Manual; ©1997; Addison Wesley

Woo, Neider, Davis; OpenGL Programming guide; ©1997; Addison Wesley

Foley, van Dam, Feiner, Hughes; Computer Graphics, Principles and Practice, ©1987; Addison Wesley

Appendix

Main Program

```
#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "Point3d.h"
#include <iostream.h>
#include<cstdlib>
#include <ctime>



HDC             hDC=NULL;           // Private GDI Device Context
HGLRC           hRC=NULL;           // Permanent Rendering Context
HWND            hWnd=NULL;          // Holds Window Handle
HINSTANCE hInstance;                // Holds Application

bool    keys[256];                  // Array Used For The Keyboard Routine
bool    active=TRUE;                // Window Active Flag Set To TRUE By Default
bool    fullscreen=FALSE;           // Fullscreen Flag Set To Fullscreen Mode By
Default

double rtri=0;
const double PI=4.0*atan(1.0);
bool test=true;
int T=0;
int c=0;

//Camera Variables
double rotx=0, roty=0;
double initx=0.0, inity=0.0, initz=0.0;
double cx=0, cy=0, cz=0;
//double ctx=0, cty=0, ctz=5.0;
double sens=0.05, rot_sens=0.5;



//Triangle Subdivision Functions and Variable Declaration
```

```
int level=6;
int index=0;
int numtri=4096;
Point3d PArray[3][4096];
Point3d CalcMidPoint(Point3d p1, Point3d p2);
void DivideMatrix(Point3d point1, Point3d point2, Point3d point3, int lvl);
void BuildDefinition();
double GetAngle(int type, int c, int d);
double GetAngle(double color);
double GetColor(double angle);
double GetDegrees(double r);
double GetRadians(double d);
double GetRoot(int c, int d);
double F(double x, double a, double b);
double D(double x, double a, double b);
GLuint DataSet;
GLvoid BuildLists();


LRESULT      CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);    //
Standard Windows application declaration




GLvoid ReSizeGLScene(GLsizei width, GLsizei height)              // Resize And
Initialize The GL Window
{
        if (height==0)
        {
                height=1;

        }

        glViewport(0,0,width,height);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        //Aspect Ratio Of The Window
        gluPerspective(45.0f,(GLdouble)width/(GLdouble)height,0.1f,100.0f);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}
```

```
int InitGL(GLvoid)
{


 glShadeModel(GL_SMOOTH);                                    // Enable Smooth
Shading
 glClearColor(0.0f, 0.14f, 0.15f, 0.5f);                  // Background Color
 glClearDepth(1.0f);                                          // Depth Buffer Setup
 glEnable(GL_DEPTH_TEST);                                    // Enables Depth Test
 glDepthFunc(GL_LEQUAL);
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);


 //Triangle Initialization
 Point3d p1, p2, p3;
 p1.Translate( 0.00, 0.00, (sqrt(3.0)/2.0) - (1.0/ (2.0*sqrt(3.0))) );
 p2.Translate((-1.0/2.0), 0.00, (-1.0/(2.0*sqrt(3.0))));
 p3.Translate(( 1.0/2.0), 0.00, (-1.0/(2.0*sqrt(3.0))));
 DivideMatrix(p1, p2, p3, level);
 BuildDefinition();
 BuildLists();

 return TRUE;
}

int DrawGLScene(GLvoid)
{
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();

 glTranslated(initx, inity, initz-6.0);

 //Camera Translation & Rotation
 glTranslated(cx,cy,cz);
 glRotated(roty, 1.0, 0.0, 0.0);
 glRotated(rotx, 0.0, 1.0, 0.0);

 //Scale and Draw Point Array
 glScaled(5.0, 0.2, 5.0);
 glPushMatrix();
 glCallList(DataSet);
 glPopMatrix();
```

```
 return TRUE;
}

/*************************************************************************
*                                                                        *
* Windows application Code Goes Here along with the killGL         *
* functions. Code has been removed because it is not very important*
* to the logic behind this program.                                *
*                                                                        *
*************************************************************************/



//Recursive Triangle Subdivision Method
void DivideMatrix(Point3d p1, Point3d p2, Point3d p3, int lvl)
{
    if (lvl > 0)
    {
    Point3d mid1_2 = CalcMidPoint(p1, p2);
    Point3d mid2_3 = CalcMidPoint(p2, p3);
    Point3d mid3_1 = CalcMidPoint(p3, p1);
    DivideMatrix(p1,     mid1_2, mid3_1, lvl - 1);
    DivideMatrix(mid1_2,     p2, mid2_3, lvl - 1);
    DivideMatrix(mid1_2, mid2_3, mid3_1, lvl - 1);
    DivideMatrix(mid3_1, mid2_3,     p3, lvl - 1);
    }
    else
    {
     PArray[0][index]=p1;
     PArray[1][index]=p2;
     PArray[2][index]=p3;
    index++;
    }


}


Point3d CalcMidPoint(Point3d p1, Point3d p2)
{
Point3d point=Point3d(0.0, 0.0, 0.0);
point.Translate( ((p1.x + p2.x)/2.0),((p1.y + p2.y)/2.0),((p1.z + p2.z)/2.0));
return point;
}
```

```
//Initializes Final List and transposes information to GL Code

GLvoid BuildLists()
{
double r, g, b;
DataSet = glGenLists(1);
glNewList(DataSet, GL_COMPILE);
index=0;
 for(index=0; index < numtri; index++)
 {

    glBegin(GL_TRIANGLES);
     glNormal3f( 0.0, 1.0, 0.0);
     glColor3d(PArray[0][index].r,PArray[0][index].g,PArray[0][index].b);
      glVertex3d(
       PArray[0][index].x,
       PArray[0][index].y,
       PArray[0][index].z);

     glColor3d(PArray[1][index].r,PArray[1][index].g,PArray[1][index].b);
      glVertex3d(
       PArray[1][index].x,
       PArray[1][index].y,
       PArray[1][index].z);

     glColor3d(PArray[2][index].r,PArray[2][index].g,PArray[2][index].b);
      glVertex3d(
       PArray[2][index].x,
       PArray[2][index].y,
       PArray[2][index].z);
    glEnd();

 }
glEndList();

}


//Defines Color and height of each Point in the Point Array
void BuildDefinition()
{
    int c,d;
    for(c=0; c < 3;c++)
    {
```

```
      for(d=0; d< numtri;d++)
      {
       PArray[c][d].r=GetColor(GetAngle(0, c, d));
       PArray[c][d].g=GetColor(GetAngle(1, c, d));
       PArray[c][d].b=GetColor(GetAngle(2, c, d));
       PArray[c][d].y=GetRoot(c, d);
      }

     }


}

//Returns one of three Angles the point represents
double GetAngle(int type, int c, int d)
{
 double angle=0.0, y=PArray[c][d].z, x=PArray[c][d].x;
 if(type==0)
 {
  angle=(360.0/sqrt(3.0)) * (y + (1.0 /(2.0*sqrt(3.0) )));
 }
 else if(type==1)
 {
  double xp = (x - (sqrt(3.0) * y) + 1.0)/4.0;
  double yp = (-1.0*sqrt(3.0)*xp) + (1.0/sqrt(3.0));
  double d = sqrt( pow(x-xp,2.0) + pow(y-yp,2.0) );
  angle = (360.0*d)/sqrt(3.0);
 }
 else if(type==2)
 {
  double xp = (x + (sqrt(3.0) * y) - 1.0)/4.0;
  double yp = (sqrt(3.0)*xp) + (1.0/sqrt(3.0));
  double d = sqrt( pow(x-xp,2.0) + pow(y-yp,2.0) );
  angle = (360.0*d)/sqrt(3.0);

 }
 return angle;
}

double GetAngle(double color)
{
 double angle = color*180.0;
 return angle;
}

double GetDegrees(double r)
```

```
{
 double d=r*(180.0/PI);
 return d;
}

double GetRadians(double d)
{
 double r=d*(PI/180.0);
 return r;
}

double GetColor(double angle)
{
 double color=angle / 180.0;
 return color;
}


//Newton's Root Function to find height of any given triangle
double GetRoot(int c, int d)
{
      double a=0.0;
      double b=0.0;
      double newguess=0.0;


 if( sqrt(pow(GetAngle(PArray[c][d].b)-60.0, 2.0)) < 0.0001
  || sqrt(pow(GetAngle(PArray[c][d].g)-60.0, 2.0)) < 0.0001
  || sqrt(pow(GetAngle(PArray[c][d].b)-60.0, 2.0)) < 0.0001 )
 {
  PArray[c][d].r=0.0;
  PArray[c][d].g=0.0;
  PArray[c][d].b=0.0;
  newguess=5;
 }
 else if( sqrt(pow(GetAngle(PArray[c][d].b)-180.0, 2.0)) < 0.0001
      || sqrt(pow(GetAngle(PArray[c][d].g)-180.0, 2.0)) < 0.0001
      || sqrt(pow(GetAngle(PArray[c][d].b)-180.0, 2.0)) < 0.0001 )
 {
  PArray[c][d].r=0.0;
  PArray[c][d].g=0.0;
  PArray[c][d].b=0.0;
  newguess=1.0;
 }
 else if(sqrt(pow(GetAngle(PArray[c][d].r),2.0)) < 0.00001
     || sqrt(pow(GetAngle(PArray[c][d].g),2.0)) < 0.00001
```

```
     || sqrt(pow(GetAngle(PArray[c][d].b),2.0)) < 0.00001 )
 {
  PArray[c][d].r=0.0;
  PArray[c][d].g=0.0;
  PArray[c][d].b=0.0;
  newguess=1.0;
 }
 else
 {

a=sin(GetRadians(GetAngle(PArray[c][d].r)))/sin(GetRadians(GetAngle(PArray[c][d].b)
));

b=sin(GetRadians(GetAngle(PArray[c][d].g)))/sin(GetRadians(GetAngle(PArray[c][d].b)
));
  PArray[c][d].A=a;
  PArray[c][d].B=b;
  double guess=2.0;
  newguess=guess - F(guess, a, b)/D(guess, a, b);
  while ( sqrt(pow(guess - newguess,2.0)) > 0.00001)
  {
   guess=newguess;
   newguess=guess - F(guess, a, b)/D(guess, a, b);
  }
 }
 if(newguess > 10.0)
 {
  newguess=10.0;
 }
 else if(newguess < -10.0)
 {
  newguess=-10.0;
 }
 return newguess;
}

//Function of modified Pythagoras Theorem
double F(double x, double a, double b)
{
 double r = 1.0 - (pow(a,x) + pow(b,x));
 return r;
}

//Derivative of modified Pythagoras Theorem
double D(double x, double a, double b)
{
```

```
 double r = -(log(a)*(pow(a,x)) + log(b)*(pow(b,x)));
 return r;
}
```

Point Class Header

```
/*******************************
*                     *
*       Point3d.h         *
*                     *
*******************************/


#ifndef POINT3D_H
#define POINT3D_H


class Point3d
{
    public:
     Point3d();
     Point3d(const Point3d & p);
     Point3d(double initX, double initY, double initZ);
     Point3d(double initX, double initY, double initZ, int initindex);
     Point3d & operator = (const Point3d &);
     double x;
     double y;
     double z;
     float r, g, b;
     double A, B;
     int index;
     void ChangeColor(float newR, float newG, float newB);
     void Translate(double newX, double newY, double newZ);
     void Translate(Point3d temp);
};
#endif
```

Point Class

```
/*******************************
*                      *
*      Point3d.cpp          *
*                      *
*******************************/



#include "Point3d.h"

Point3d::Point3d():
    x(0.0),
    y(0.0),
    z(0.0),
    index(0),
    r(1.0),
    g(1.0),
    b(1.0),
    A(1.0),
    B(1.0)
{

};


Point3d::Point3d(double initX, double initY, double initZ):
    x(initX),
    y(initY),
    z(initZ),
    index(0),
    r(1.0),
    g(1.0),
    b(1.0),
    A(1.0),
    B(1.0)
{

};

Point3d::Point3d(double initX, double initY, double initZ, int initindex):
    x(initX),
    y(initY),
    z(initZ),
    index(initindex),
```

```
    r(1.0),
    g(1.0),
    b(1.0),
    A(1.0),
    B(1.0)
{

};

Point3d::Point3d(const Point3d & p)
{
 x=p.x;
 y=p.y;
 z=p.z;
 r=p.r;
 g=p.g;
 b=p.b;
 A=p.A;
 B=p.B;
}

Point3d & Point3d::operator = (const Point3d & p)
{
 x=p.x;
 y=p.y;
 z=p.z;
 r=p.r;
 g=p.g;
 b=p.b;
 A=p.A;
 B=p.B;

 return *this;
}

void Point3d::ChangeColor(float newR, float newG, float newB)
{
 r=newR;
 g=newG;
 b=newB;
};

void Point3d::Translate(double newX, double newY, double newZ)
{
 x=newX;
 y=newY;
```

```
 z=newZ;
}

void Point3d::Translate(Point3d temp)
{
 x=temp.x;
 y=temp.y ;
 z=temp.z;
}
```