**Scaling and Translations**

By John Ying

Introduction

In many different games, the size or translation aka position will need to be updated. The scaling and translations program provides the update needed for those processes. To provide an example of real game use, say that Mario suddenly got a power up mushroom, his size (scaling) would need to be readjusted to the correct amount as well as the fact that he should look like he grew from the same spot, not grew forward. The program also provides the scaling and translations separated as well as together and about a center.

Methods

While the main().cpp file is highly important and integral to the full use of the functions, all that is really needed is to provide the user with the correct input and output as well as the correct order to use the functions. With this, the focus of methods will be upon the scaling and translations functions within the vector class.

Within the translations method, the user provided the original points to be transformed as well as the translation amount. Mathematically, the matrix multiplication is a lot larger but it can be reduced down to the below arithmetic by eliminating the unnecessary amount of zero products.

x += xT;

y += yT;

z += zT;

vPoint = wT;

With the scaling method, the user provided the original points to be transformed as well as the scaling amount. Much similar to the translation above, many sets of math can be eliminated since the product of zero’s still turn out to be zero.

x \*= xS;

y \*= yS;

z \*= zS;

vPoint \*= wS;

However, when scaling about a center, to allow an ease of use in processes, the concatenation needed to first be found. With this, I created the crossProduct4 method. With user provided scaling and center points, the concatenation was founded by the code below:

x = xC \* (1 - xS);

y = yC \* (1 - yS);

z = zC \* (1 - zS);

vPoint = wS;

Once the cross product concatenation was found, those points could be used against the originally given points by the user to find the scaling about a center.

x = (x \* xS) + (concat.x \* wS);

y = (y \* yS) + (concat.y \* wS);

z = (z \* zS) + (concat.z \* wS);

vPoint = wS;

Results

Figuring out a logical test center when scaling about one was out of my ability with the points I had given myself so the calculations are realistically not real but the calculations were correct and are projected finally in the transformation column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Original Points | Scaling | Translation | Scale about Center | Center of Center | Transformation |
| 0, 1, 5, 1 | 0,2,15,1 | 4,7,4,1 |  |  | 0,1,14,1 |
| 5, 2, 3, 1 | 5,4,9,1 | 9,8,2,1 |  |  | 5,3,6,1 |
| 4, 2, -6, 1 | 4,4,-18,1 | 8,8,-7,1 |  |  | 4,3,-30,1 |
| 4, -2, 4, 1 | 4,-4,12,1 | 8,4,3,1 |  |  | 4,-5,10,1 |
|  | 1, 2, 3, 1 | 4, 6, -1, 1 | 1, 2, 4, 1 | .5, 1, 2, 1 |  |

Conclusion

This program calculates the scaling, translation, and scaling about a center depending on what the user requested based upon their input. Everything should function perfectly fine as long as the numbers input do not exceed 7 as they are calculated by float.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/

/ scalingAndTranslations.cpp provides the main source for taking

/ user provided input of points on a shape, and allows the user to

/ input the correct commands to find the translation, scaling, or

/ scaling about a center of the provided shape.

/

/ John Ying, Apr 2012

/ Johnson County Community College

/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <string>

#include <cmath>

#include "VectorClass.h"

using namespace::std;

int main()

{

float x, y, z, vPoint; //provides place holders for the user provided input

Vector3D shape1[4]; //starts the necessary vector array that makes up a shape

char repeatCheck = 'y'; //place holder check to see if the user wishes to do another transformation

//asks the user for all of the coordinate points of the shape

for (int i = 0; i < 4; i++)

{

cout << "Please input the coordinates for point " << i << ": {x, y, z, w}" << endl;

cin >> x >> y >> z >> vPoint;

shape1[i].setRectGivenRect(x, y, z, vPoint);

}

while( repeatCheck == 'y')//while the program should be done again

{

string trans;//place holder for what type of transformation is to be done

//scaling or translating variables

float x = 1;

float y = 1;

float z = 1;

float w = 1;

//scaling about the center (center points)

float xC = 1;

float yC = 1;

float zC = 1;

cout << "What kind of transformation would you like to do?\n";

cout << "t = translation" << endl << "rs = raw scaling" << endl << "sc = scaling about a center" << endl;

cin >> trans;

if ( trans == "t" || trans == "rs" || trans == "sc" )//checks if the input transformation is a valid input

{

if ( trans == "t")//if translation is chosen

{

cout << "What translations would you like to do? {x y z w}" << endl;

cin >> x >> y >> z >> w;

for (int count = 0; count < 4; count++)//changes the points of the shape based on translation

{

shape1[count].translation(x, y, z, w);

}

}

else if (trans == "rs")//if raw scaling is chosen

{

cout << "What scaling would you like to do? {x y z w}" << endl;

cin >> x >> y >> z >> w;

for (int count = 0; count < 4; count++)//changes the points of the shape based on scaling

{

shape1[count].rawScaling(x, y, z, w);

}

}

else if (trans == "sc")//if scaling about a center is chosen

{

Vector3D concat;//holder for the cross product to find the concatenation

cout << "What scaling do you wish to do? {x y z w}" << endl;

cin >> x >> y >> z >> w;

cout << "What is the center of the object? {x y z}" << endl;

cin >> xC >> yC >> zC;

concat.crossProduct4(x, y, z, w, xC, yC, zC);//finds the concatenation based on the scalar and center points

for (int count = 0; count < 4; count++)//changes the points of the shape based on given concatenation and scaling points

{

shape1[count].scalingCenter(concat, x, y, z, w);

}

}

}

else //if the user input an incorrect transformation answer

{

cout << "Incorrect transformation input.\n";

}

//provides the final transformation coordinates by looping through each point in the shape by array

cout << "The transformation output is:\n";

for (int c = 0; c < 4; c++)

{

shape1[c].printRect();

}

//asks the user if they would like another transformation done on the newly transformed coordinates

cout << "Would you like to do another transformation? y = yes; n = no" << endl;

cin >> repeatCheck;

//tells the user that the program will end since they input no

if (repeatCheck == 'n')

{

cout << "Closing program....\n";

}

}

return 0;//returning an int allows for use of executable programs requiring a return on main

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/

/ VectorClass.h sets the starting point for many different types

/ of calculations ranging from projections, predictions, and any

/ other type of calculations based upon vectors in 2D or 3D space.

/

/ John Ying, Mar 2012

/ Johnson County Community College

/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <string>

#include <cmath>

using namespace::std;

class Vector3D //the Vector3D class declaration and beginning

{

public:

Vector3D(void) // constructor for the class Vector3D

{

x = 0.0;

y = 0.0;

z = 0.0;

vPoint = 0.0;

//executed when Vector3D is created

}

void setRectGivenRect( float setX, float setY, float setZ, float setvPoint ) //sets the x, y, and z variables in the private to the given parameters

{

x = setX;

y = setY;

z = setZ;

vPoint = setvPoint;

}

void setRectGivenPolar( float mag, float degrees ) //sets the x and y based upon a given magnitude and degrees

{

x = mag\*cos(degrees);

y = mag\*sin(degrees);

}

void setRectGivenMagHeadPitch( float mag, float heading, float pitch) //sets the x, y, and z based upon a given magnitude, heading, and pitch

{

x = mag\*cos(pitch)\*cos(heading);

y = mag\*cos(pitch)\*sin(heading);

z = mag\*sin(pitch);

}

void printRect(void) //public function for the class to print the private x, y, and z variables

{

cout << endl;

cout << "x: " << x << endl;

cout << "y: " << y << endl;

cout << "z: " << z << endl;

cout << "vPoint(w): " << vPoint << endl;

}

void printMagHeadPitch() //public function for the class to print the magnitude, heading, and pitch

{

cout << endl;

cout << "Magnitude: " << getMagnitude() << endl;

cout << "Heading: " << getHeading() << endl;

cout << "Pitch: " << getPitch() << endl;

}

void printAngles(void) //public function for the class to print the alpha, beta, and gamma angles

{

cout << endl;

cout << "The angles are\n";

cout << "Alpha: " << getAlpha() << endl;

cout << "Beta: " << getBeta() << endl;

cout << "Gamma: " << getGamma() << endl;

}

float getMagnitude() //returns the magnitude based on the given x, y, and z coordinates in the class

{

return (sqrt((x\*x)+(y\*y)+(z\*z)));

}

float getPitch() //returns the pitch based on the given x, y, and z coordinates in the class

{

if (getMagnitude() == 0)

return 0;

else

return (asin(z/getMagnitude()));

}

float getHeading() //returns the heading based on the given x, y, and z coordinates in the class

{

if (sqrt((x\*x)+(y\*y)) == 0)

return 0;

else

return (acos(x/(sqrt((x\*x)+(y\*y)))));

}

float getAlpha()//returns the alpha angle based on the given x, y, and z coordinates in the class

{

if (getMagnitude() == 0)

return 0;

else

return (acos(x / getMagnitude()));

}

float getBeta()//returns the beta angle based on the given x, y, and z coordinates in the class

{

if (getMagnitude() == 0)

return 0;

else

return (acos(y / getMagnitude()));

}

float getGamma()//returns the gamma angle based on the given x, y, and z coordinates in the class

{

if (getMagnitude() == 0)

return 0;

else

return (acos(z / getMagnitude()));

}

float getX() //returns the x value in the class

{

return x;

}

float getY() //returns the y value in the class

{

return y;

}

float getZ() //returns the z value in the class

{

return z;

}

Vector3D projection(Vector3D v, Vector3D u) //this finds the projection of v onto u.

{

//variable declarations

float topAdd;

float botAdd;

float ans;

Vector3D proj;

//function math

topAdd = (u.x \* v.x) + (u.y \* v.y) + (u.z \* v.z);

botAdd = (v.x \* v.x) + (v.y \* v.y) + (v.z \* v.z);

ans = topAdd / botAdd;

proj.x = ans \* v.x;

proj.y = ans \* v.y;

proj.z = ans \* v.z;

return proj; //returns the vector answer

}

void crossProduct( Vector3D v2, Vector3D v3 ) //finds the cross product of the given parameter vectors into the current vector

{

x = (v2.y \* v3.z) - (v2.z \* v3.y);

y = (v2.z \* v3.x) - (v2.x \* v3.z);

z = (v2.x \* v3.y) - (v2.y \* v3.x);

}

void closestPointLine ( const Vector3D ship, const Vector3D point, const Vector3D direction ) //ship is the random point outside, point and direction are the two points of the line(PQ)

{

//x,y, z is point aka toms ship

//sighted at position point in the direction of (direction)

//variable declarations

Vector3D pq;

Vector3D proj;

//solves for PQ by doing Q - P(point) in vector projections

pq.x = ship.x - point.x;

pq.y = ship.y - point.y;

pq.z = ship.z - point.z;

proj = projection(direction, pq); //solves and stores the projection of the given parameters (v, u)

x = point.x + proj.x;

y = point.y + proj.y;

z = point.z + proj.z;

}

void closestPointPlane (Vector3D ship, Vector3D planeP1, Vector3D planeP2, Vector3D planeP3) //solves for the closest point by a given plane and Q(ship) coordinate

{

//variable declarations

Vector3D proj; //vector projection storage

Vector3D normal; //normal vector storage

Vector3D v1; //vector 1

Vector3D v2; //vector 2

Vector3D point; //solving for P(point)

//solves for the needed cross product vectors based upon the 3 given points

v1.x = planeP1.x - planeP2.x;

v1.y = planeP1.y - planeP2.y;

v1.z = planeP1.z - planeP2.z;

v2.x = planeP1.x - planeP3.x;

v2.y = planeP1.y - planeP3.y;

v2.z = planeP1.z - planeP3.z;

normal.crossProduct (v1, v2); //finds the cross product of two vectors

//solves for and finds the P(point) by a randomly given point and Q(ship)

point.x = ship.x - planeP1.x;

point.y = ship.y - planeP1.y;

point.z = ship.z - planeP1.z;

proj = projection(normal, point); //solves for and stores the projection of the normal vector onto point

//solves for and stores the variables for the S in the projection of the closest point from a plane

x = ship.x - proj.x;

y = ship.y - proj.y;

z = ship.z - proj.z;

}

void translation(float xT, float yT, float zT, float wT) //translates the current vector by the given parameter points

{

x += xT;

y += yT;

z += zT;

vPoint = wT;

}

void rawScaling(float xS, float yS, float zS, float wS) //scales the current vector by the given paramter points

{

x \*= xS;

y \*= yS;

z \*= zS;

vPoint \*= wS;

}

void scalingCenter(Vector3D concat, float xS, float yS, float zS, float wS) //multiplies the concatenation and the current vector to find the scaling about a center

{

x = (x \* xS) + (concat.x \* wS);

y = (y \* yS) + (concat.y \* wS);

z = (z \* zS) + (concat.z \* wS);

vPoint = wS;

}

void crossProduct4( float xS, float yS, float zS, float wS, float xC, float yC, float zC ) //finds the concatenation of the two points

{

x = xC \* (1 - xS);

y = yC \* (1 - yS);

z = zC \* (1 - zS);

vPoint = wS;

}

private:

float x; //Private Variable Declarations

float y;

float z;

float vPoint;

};

**Post Lab Questions**

1. A centimeter is a 100th of a meter 0.01. If the spaceship has a length of 900m, then the most exact amount may be up to 5 digits per example 865.31m or 86,631cm. There are two extra significant digits because a float holds 7 and the spaceship parameters only use up to 5. If an error of 1 unit in my last digit is caused, the minimum number of multiplications needed is 100. 100 because that would cause the 1 centimeter difference. At 30 frames a second, errors would be noticeable every 4 seconds because the error would have occurred at 120 frames if the calculation was every frame.
2. The approach would help avoid degredation by not allowing any room for the object to be mis-translated. Since the size of objects will usually stay the same, the largest problem is the calculation of distance. If the correct distance is solved first before the chance for error or movement, then there is no degredation.