q7

Thursday, May 02, 2024 9:09 PM

7) Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it.

```
//==================ROTATION 3D PERSPECTIVE==========================

#include <iostream>
#include <conio.h>
#include <math.h>


#include "C:\Users\akash\OneDrive\Desktop\26th_april\dda.cpp"
//#include "C:\Users\akash\OneDrive\Desktop\26th_april\dda.cpp"


#include <vector>
#include <utility>


 #include <graphics.h>
using namespace std;


double degreesToRadians(double degrees) {
   return degrees * M_PI / 180.0;
}


class my_matrix{
   public:
      double** matrix ;
      int row;
      int col;


   my_matrix(){


   }


   my_matrix(int r,int v){


      row = r;
      col = 4; // for homogeneous


      matrix = new double*[r];


       // Allocate memory for each row
         for (int i = 0; i < r; ++i) {
            matrix[i] = new double[col]; // Create an array of integers for each row
         }


           // Initialize each element to 1
         for (int i = 0; i < r; ++i) {
            for (int j = 0; j < col; ++j) {
               matrix[i][j] = v;
            }
         }
   }
```

```
    void print_obj(){


  cout<<"-------------------------------------OBJECT-------------------------
------------------\n";
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
    cout<<"-------------------------------------------------------------------------
------------------\n";
    }


    void deallocate_matrix(){
         // Free dynamically allocated memory
    for (int i = 0; i < row; ++i) {
        delete[] matrix[i];
    }
    delete[] matrix;
    }
};




void create_scaling_matrix(int sx,int sy,int sz, my_matrix *obj){
    obj->matrix[0][0] = sx;
    obj->matrix[1][1] = sy;
    obj->matrix[2][2] = sz;
    obj->matrix[3][3] = 1;


}


void create_rotation_matrix_x_axis(double angle, my_matrix *obj){  //
x_axis


double radians = degreesToRadians(angle);
 obj->matrix[0][0] = 1;
 obj->matrix[3][3] = 1;


 obj->matrix[1][1] = cos(radians);
 obj->matrix[1][2] = sin(radians);


 obj->matrix[2][2] = cos(radians);
 obj->matrix[2][1] = -1*sin(radians);


}






void create_perspective_matrix(double angle, double zc , my_matrix
*obj){


    double radians = degreesToRadians(angle);

    obj->matrix[0][0] = cos(radians);
    obj->matrix[0][3] = sin(radians)/zc;


    obj->matrix[1][1] = 1;
```

```
    obj->matrix[2][2] = 0;
    obj->matrix[3][3] = 1;




    obj->matrix[3][1] = 2;


    obj->matrix[2][0] = sin(radians);
    obj->matrix[2][3] = (-1*cos(radians))/zc;


}




void matrix_multiplication(my_matrix *obj , my_matrix *scaling ,
my_matrix *result){
     // Function to perform matrix multiplication


       if (obj->col != scaling->row) {
          std::cerr << "Error: Matrix dimensions mismatch for
multiplication." << std::endl;
          exit(1);
       }
       for (int i = 0; i < obj->row; ++i) {
          for (int j = 0; j < scaling ->col; ++j) {
             result -> matrix[i][j] = 0;


             for (int k = 0; k < obj -> col ; ++k) {
                result -> matrix[i][j] +=  obj->matrix[i][k] * scaling -> matrix[k]
[j];
             }


          }
       }

}









void draw_object(my_matrix *obj,int colorr){   // only for 6 faces
    int x1=0;
    int y1=0;


    int x2=0;
    int y2=0;
    x1 = obj->matrix[0][0];
    y1 = obj->matrix[0][1];


    vector< pair< double, double> > points;
    vector< pair< double, double> > points2;


    for(int i=0;i< 4;i++){
       points.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));
       if(i==3){
          points.push_back(make_pair(obj->matrix[0][0],obj->matrix[0][1]));
       }
    }


     for(int i=4;i< 8;i++){
```

```
        points2.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));


    }


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


        //print  LINE 1
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points[i+1].first<< " , "<<points[i+1].second<<"   : : ";
        delay(1000);
     dda(points[i].first,   points[i].second   , points[i+1].first,
points[i+1].second , colorr );


        cout<<endl;
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points2[i].first<< " , "<<points2[i].second<<"   : : ";


      delay(1000);
       dda(points[i].first , points[i].second,  points2[i].first ,
points2[i].second, colorr  );


        cout<<"\n----------------------------------------------------------"<<endl;
    }




   points2.push_back(make_pair(points2[0].first,points2[0].second));


    for(int i=0;i<points2.size()-1;i++){
        cout<<points2[i].first<< " , "<<points2[i].second<<"   : : "
<<points2[i+1].first<< " , "<<round(points2[i+1].second)<<"   : : \n";
        dda( points2[i].first  ,points2[i].second    , round( points2[i+1].first )  ,
round(points2[i+1].second) , colorr);
         delay(1000);
    }
}









inline double clamp(double x, double epsilon = 1e-6) {
    return (x < epsilon && x > -epsilon) ? 0.0 : x;
}


void making_last_column_one(my_matrix *obj,int row){


    for(int i =0; i<row ;i++ ){
        obj->matrix[i][0] =  clamp( (obj->matrix[i][0] / obj->matrix[i][3]));
        obj->matrix[i][1] =  clamp( (obj->matrix[i][1] / obj->matrix[i][3]) );
        obj->matrix[i][2] = clamp(  (obj->matrix[i][2] / obj->matrix[i][3])  );
        obj->matrix[i][3] =  clamp( (obj->matrix[i][3] / obj->matrix[i][3]) );
    }
}




int main(){


    int r = 8 ;
    my_matrix *obj;
```

```
if(r>=1){
    cout<<"matrix made"<<endl;
    obj  = new my_matrix(r,1);


}else{
    cout<<"invalid input";
    return 0;
}




    obj->matrix[0][0] = 0;
    obj->matrix[0][1] = 0;
    obj->matrix[0][2] = 1;


    obj->matrix[1][0] = 1;
    obj->matrix[1][1] = 0;
    obj->matrix[1][2] = 1;


    obj->matrix[2][0] = 1;
    obj->matrix[2][1] = 1;
    obj->matrix[2][2] = 1;


    obj->matrix[3][0] = 0;
    obj->matrix[3][1] = 1;
    obj->matrix[3][2] = 1;


    obj->matrix[4][0] = 0;
    obj->matrix[4][1] = 0;
    obj->matrix[4][2] = 0;


    obj->matrix[5][0] = 1;
    obj->matrix[5][1] = 0;
    obj->matrix[5][2] = 0;


    obj->matrix[6][0] = 1;
    obj->matrix[6][1] = 1;
    obj->matrix[6][2] = 0;


    obj->matrix[7][0] = 0;
    obj->matrix[7][1] = 1;
    obj->matrix[7][2] = 0;




 cout<<"\n object
==========================================================
==========\n";
    obj->print_obj();




my_matrix *rotation_matrix = new my_matrix(4,0);
create_rotation_matrix_x_axis(45.0,rotation_matrix);


    cout<<"\n rotation matrix
==========================================================
==========\n";
    rotation_matrix->print_obj();
```

```
// rotating object by multiplying
my_matrix *result_after_rotation =  new my_matrix(r,0);
matrix_multiplication(obj, rotation_matrix , result_after_rotation);


// write code for rotation



my_matrix *perspective_projection_matrix = new my_matrix(4,0);
create_perspective_matrix(60,2.5,perspective_projection_matrix);


cout<<"\nperspective matrix() :"<<endl;
perspective_projection_matrix->print_obj();


my_matrix *result_after_projection =  new my_matrix(r,0);


matrix_multiplication(result_after_rotation ,
perspective_projection_matrix , result_after_projection);


making_last_column_one(result_after_projection,r);



cout<<"result after projection\n";
result_after_projection->print_obj();


    int gd = DETECT, gm;
   char pathtodriver[] = "";
   initgraph(&gd, &gm, pathtodriver);



 //  draw_object(result2,GREEN);
 //   delay(2000);



// scaling object by multiplying
my_matrix *scaling_mat =  new my_matrix(4,0);
create_scaling_matrix(100,100,0,scaling_mat);


my_matrix *scaled_obj =  new my_matrix(r,0);
matrix_multiplication(result_after_projection , scaling_mat,  scaled_obj );


draw_object(scaled_obj ,BLUE);



getch();
closegraph();


return 0;
}
```

```
//===============.
Scaling============================


// 7) Write a program to apply various 3D transformations on a 3D object
and then apply parallel
// and perspective projection on it.
// Write a program to apply various 3D transformations on a 3D object
and then apply parallel
// and perspective projection on it.



//  2 point perspective view
//  rotating 1 point perspective gives --> 2 point perspective


#include <iostream>
#include <conio.h>
#include <math.h>


#include "C:\Users\akash\OneDrive\Desktop\26th_april\dda.cpp"


#include <vector>
#include <utility>


// #include <graphics.h>
using namespace std;


class my_matrix{
   public:
      double** matrix ;
      int row;
      int col;


   my_matrix(){


   }


   my_matrix(int r,int v){


      row = r;
      col = 4; // for homogeneous


      matrix = new double*[r];


        // Allocate memory for each row
         for (int i = 0; i < r; ++i) {
            matrix[i] = new double[col]; // Create an array of integers for
each row
         }


           // Initialize each element to 1
         for (int i = 0; i < r; ++i) {
            for (int j = 0; j < col; ++j) {
               matrix[i][j] = v;
            }
         }
   }
```

```
    void print_obj(){


  cout<<"-------------------------------------OBJECT-------------------------
------------------\n";
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
    cout<<"-------------------------------------------------------------------------
------------------\n";
    }


    void deallocate_matrix(){
         // Free dynamically allocated memory
    for (int i = 0; i < row; ++i) {
        delete[] matrix[i];
    }
    delete[] matrix;
    }
};




void create_scaling_matrix(int sx,int sy,int sz, my_matrix *obj){
    obj->matrix[0][0] = sx;
    obj->matrix[1][1] = sy;
    obj->matrix[2][2] = sz;
    obj->matrix[3][3] = 1;


}


void create_perspective_matrix(double p,double yc,double zc,
my_matrix *obj){
    // obj->matrix[0][3] = p;
    // obj->matrix[1][3] = -1*(1/yc);
    // obj->matrix[2][3] = -1*(1/zc);
    // obj->matrix[3][3] = 1;
    // obj->matrix[2][2] = 0;


    // obj->matrix[0][0] = 1;
    // obj->matrix[1][1] = 1;

    obj->matrix[0][0] = 0.5;
    obj->matrix[1][1] = 1;
    obj->matrix[2][2] = 0;
    obj->matrix[3][3] = 1;


    obj->matrix[2][0] = 0.866;
    obj->matrix[3][1] = 2;
    obj->matrix[3][0] = 0 ;


    obj->matrix[0][3] = 0.346 ;
    obj->matrix[2][3] = -0.2 ;


}




void matrix_multiplication(my_matrix *obj , my_matrix *scaling ,
my_matrix *result){
```

```
    // Function to perform matrix multiplication


    if (obj->col != scaling->row) {
        std::cerr << "Error: Matrix dimensions mismatch for
multiplication." << std::endl;
        exit(1);
    }
    for (int i = 0; i < obj->row; ++i) {
        for (int j = 0; j < scaling ->col; ++j) {
            result -> matrix[i][j] = 0;


            for (int k = 0; k < obj -> col ; ++k) {
                result -> matrix[i][j] +=  obj->matrix[i][k] * scaling -> matrix[k]
[j];
            }


        }
    }

}









void draw_object(my_matrix *obj,int colorr){   // only for 6 faces
   int x1=0;
   int y1=0;


   int x2=0;
   int y2=0;
   x1 = obj->matrix[0][0];
   y1 = obj->matrix[0][1];


   vector< pair< double, double> > points;
   vector< pair< double, double> > points2;


   for(int i=0;i< 4;i++){
      points.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));
      if(i==3){
         points.push_back(make_pair(obj->matrix[0][0],obj->matrix[0][1]));
      }
   }


    for(int i=4;i< 8;i++){
      points2.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));
      // if(i==3){
      //     points2.push_back(make_pair(obj->matrix[0][0],obj->matrix[0]
[1]));
      // }
   }


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


      //print  LINE 1
      cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points[i+1].first<< " , "<<points[i+1].second<<"   : : ";
      delay(1000);
      dda(points[i].first,   points[i].second   , points[i+1].first,
points[i+1].second , colorr );


      cout<<endl;
```

```
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points2[i].first<< " , "<<points2[i].second<<"   : : ";


         delay(1000);
        dda(points[i].first , points[i].second,  points2[i].first ,
points2[i].second, colorr  );


        cout<<"\n----------------------------------------------------------"<<endl;
    }




  // points2.push_back(make_pair(obj->matrix[0][0],obj->matrix[0][1]));


    for(int i=0;i<points2.size();i++){
        cout<<points2[i].first<< " , "<<points2[i].second<<"   : : "
<<points2[i+1].first<< " , "<<round(points2[i+1].second)<<"   : : \n";
        dda( points2[i].first  ,points2[i].second    , points2[i+1].first   ,
round(points2[i+1].second) , colorr);
         delay(1000);
    }
}









inline double clamp(double x, double epsilon = 1e-6) {
    return (x < epsilon && x > -epsilon) ? 0.0 : x;
}


void making_last_column_one(my_matrix *obj,int row){


    for(int i =0; i<row ;i++ ){
        obj->matrix[i][0] =  clamp( (obj->matrix[i][0] / obj->matrix[i][3]));
        obj->matrix[i][1] =  clamp( (obj->matrix[i][1] / obj->matrix[i][3]) );
        obj->matrix[i][2] = clamp(  (obj->matrix[i][2] / obj->matrix[i][3])  );
        obj->matrix[i][3] =  clamp( (obj->matrix[i][3] / obj->matrix[i][3]) );
    }
}




int main(){


    int r = 8 ;
    my_matrix *obj;


if(r>=1){
    cout<<"matrix made"<<endl;
    obj  = new my_matrix(r,1);
    obj->print_obj();
}else{
    cout<<"invalid input";
    return 0;
}




    obj->matrix[0][0] = 0;
    obj->matrix[0][1] = 0;
    obj->matrix[0][2] = 1;
```

```
    obj->matrix[1][0] = 1;
    obj->matrix[1][1] = 0;
    obj->matrix[1][2] = 1;


    obj->matrix[2][0] = 1;
    obj->matrix[2][1] = 1;
    obj->matrix[2][2] = 1;


    obj->matrix[3][0] = 0;
    obj->matrix[3][1] = 1;
    obj->matrix[3][2] = 1;


    obj->matrix[4][0] = 0;
    obj->matrix[4][1] = 0;
    obj->matrix[4][2] = 0;


    obj->matrix[5][0] = 1;
    obj->matrix[5][1] = 0;
    obj->matrix[5][2] = 0;


    obj->matrix[6][0] = 1;
    obj->matrix[6][1] = 1;
    obj->matrix[6][2] = 0;


    obj->matrix[7][0] = 0;
    obj->matrix[7][1] = 1;
    obj->matrix[7][2] = 0;




int sx = 100 , sx2 = 50;
int sy = 100 , sy2 =50;
int sz = 0 , sz2 = 0;




my_matrix *scaling_matrix = new my_matrix(4,0);;
create_scaling_matrix(sx,sy,sz,scaling_matrix);


my_matrix *scaling_matrix2 = new my_matrix(4,0);;
create_scaling_matrix(sx2,sy2,sz2,scaling_matrix2);


my_matrix *perspective_projection_matrix = new my_matrix(4,0);
create_perspective_matrix(0,1000,1000
,perspective_projection_matrix);




cout<<"scaling matrix:"<<endl;
scaling_matrix->print_obj();


cout<<"\nperspective matrix() :"<<endl;
perspective_projection_matrix->print_obj();


my_matrix *result_after_projection =  new my_matrix(r,0);
matrix_multiplication(obj, perspective_projection_matrix ,
result_after_projection);
```

```
making_last_column_one(result_after_projection,r);


    int gd = DETECT, gm;
   char pathtodriver[] = "";
   initgraph(&gd, &gm, pathtodriver);



my_matrix *result2 =  new my_matrix(r,0);  // 8  x 4     4 x 4 -->  8 x 4
matrix_multiplication(result_after_projection ,scaling_matrix2,result2);


    draw_object(result2,GREEN);


    delay(2000);


// increasing scale


my_matrix *result =  new my_matrix(r,0);  // 8  x 4     4 x 4 -->  8 x 4
matrix_multiplication(result_after_projection,scaling_matrix,result);


 draw_object(result,BLUE);



getch();
closegraph();


return 0;
}


//==============. Shearing=================



// 7) Write a program to apply various 3D transformations on a 3D object
and then apply parallel
// and perspective projection on it.
// Write a program to apply various 3D transformations on a 3D object
and then apply parallel
// and perspective projection on it.



//  2 point perspective view
//  rotating 1 point perspective gives --> 2 point perspective



#include <iostream>
#include <conio.h>
#include <math.h>


#include "C:\Users\akash\OneDrive\Desktop\26th_april\dda.cpp"
//#include "C:\Users\akash\OneDrive\Desktop\26th_april\dda.cpp"


#include <vector>
#include <utility>


 #include <graphics.h>
```

```
using namespace std;


double degreesToRadians(double degrees) {
    return degrees * M_PI / 180.0;
}


class my_matrix{
    public:
        double** matrix ;
        int row;
        int col;


    my_matrix(){


    }


    my_matrix(int r,int v){


        row = r;
        col = 4; // for homogeneous


        matrix = new double*[r];


          // Allocate memory for each row
            for (int i = 0; i < r; ++i) {
                matrix[i] = new double[col]; // Create an array of integers for
each row
            }


              // Initialize each element to 1
            for (int i = 0; i < r; ++i) {
                for (int j = 0; j < col; ++j) {
                    matrix[i][j] = v;
                }
            }
    }


    void print_obj(){


  cout<<"---------------------------------------OBJECT-------------------------
------------------\n";
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
    cout<<"-----------------------------------------------------------------------------
------------------\n";
    }


    void deallocate_matrix(){
         // Free dynamically allocated memory
    for (int i = 0; i < row; ++i) {
        delete[] matrix[i];
    }
    delete[] matrix;
    }
};
```

```
void create_shearing_matrix(double b, double c, double d, double f,
double g, double i, my_matrix *obj){
    obj->matrix[0][0] = sx;
    obj->matrix[1][1] = sy;
    obj->matrix[2][2] = sz;
    obj->matrix[3][3] = 1;
}




void create_scaling_matrix(int sx,int sy,int sz, my_matrix *obj){
    obj->matrix[0][0] = sx;
    obj->matrix[1][1] = sy;
    obj->matrix[2][2] = sz;
    obj->matrix[3][3] = 1;
}


void create_perspective_matrix(double angle, double zc , my_matrix
*obj){



    double radians = degreesToRadians(angle);


    obj->matrix[0][0] = cos(radians);
    obj->matrix[0][3] = sin(radians)/zc;



    obj->matrix[1][1] = 1;
    obj->matrix[2][2] = 0;
    obj->matrix[3][3] = 1;



    obj->matrix[3][1] = 2;



    obj->matrix[2][0] = sin(radians);
    obj->matrix[2][3] = (-1*cos(radians))/zc;



}




void matrix_multiplication(my_matrix *obj , my_matrix *scaling ,
my_matrix *result){
     // Function to perform matrix multiplication



       if (obj->col != scaling->row) {
          std::cerr << "Error: Matrix dimensions mismatch for
multiplication." << std::endl;
          exit(1);
       }
       for (int i = 0; i < obj->row; ++i) {
          for (int j = 0; j < scaling ->col; ++j) {
             result -> matrix[i][j] = 0;



             for (int k = 0; k < obj -> col ; ++k) {
                result -> matrix[i][j] +=  obj->matrix[i][k] * scaling -> matrix[k]
[j];
             }



          }
       }


}
```

```
void draw_object(my_matrix *obj,int colorr){   // only for 6 faces
    int x1=0;
    int y1=0;


    int x2=0;
    int y2=0;
    x1 = obj->matrix[0][0];
    y1 = obj->matrix[0][1];


    vector< pair< double, double> > points;
    vector< pair< double, double> > points2;


    for(int i=0;i< 4;i++){
        points.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));
        if(i==3){
            points.push_back(make_pair(obj->matrix[0][0],obj->matrix[0][1]));
        }
    }


     for(int i=4;i< 8;i++){
        points2.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));


    }


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


        //print  LINE 1
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points[i+1].first<< " , "<<points[i+1].second<<"   : : ";
        delay(1000);
      dda(points[i].first,   points[i].second   , points[i+1].first,
points[i+1].second , colorr );


        cout<<endl;
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points2[i].first<< " , "<<points2[i].second<<"   : : ";


       delay(1000);
       dda(points[i].first , points[i].second,  points2[i].first ,
points2[i].second, colorr  );


        cout<<"\n----------------------------------------------------------"<<endl;
    }




   points2.push_back(make_pair(points2[0].first,points2[0].second));


    for(int i=0;i<points2.size()-1;i++){
        cout<<points2[i].first<< " , "<<points2[i].second<<"   : : "
<<points2[i+1].first<< " , "<<round(points2[i+1].second)<<"   : : \n";
        dda( points2[i].first  ,points2[i].second    , round( points2[i+1].first )  ,
round(points2[i+1].second) , colorr);
         delay(1000);
    }
}
```

```
inline double clamp(double x, double epsilon = 1e-6) {
    return (x < epsilon && x > -epsilon) ? 0.0 : x;
}


void making_last_column_one(my_matrix *obj,int row){


    for(int i =0; i<row ;i++ ){
        obj->matrix[i][0] =  clamp( (obj->matrix[i][0] / obj->matrix[i][3]));
        obj->matrix[i][1] =  clamp( (obj->matrix[i][1] / obj->matrix[i][3]) );
        obj->matrix[i][2] = clamp(  (obj->matrix[i][2] / obj->matrix[i][3])  );
        obj->matrix[i][3] =  clamp( (obj->matrix[i][3] / obj->matrix[i][3]) );
    }
}




int main(){


    int r = 8 ;
    my_matrix *obj;


if(r>=1){
    cout<<"matrix made"<<endl;
    obj  = new my_matrix(r,1);


}else{
    cout<<"invalid input";
    return 0;
}




    obj->matrix[0][0] = 0;
    obj->matrix[0][1] = 0;
    obj->matrix[0][2] = 1;


    obj->matrix[1][0] = 1;
    obj->matrix[1][1] = 0;
    obj->matrix[1][2] = 1;


    obj->matrix[2][0] = 1;
    obj->matrix[2][1] = 1;
    obj->matrix[2][2] = 1;


    obj->matrix[3][0] = 0;
    obj->matrix[3][1] = 1;
    obj->matrix[3][2] = 1;


    obj->matrix[4][0] = 0;
    obj->matrix[4][1] = 0;
    obj->matrix[4][2] = 0;


    obj->matrix[5][0] = 1;
    obj->matrix[5][1] = 0;
    obj->matrix[5][2] = 0;


    obj->matrix[6][0] = 1;
```

```
    obj->matrix[6][1] = 1;
    obj->matrix[6][2] = 0;


    obj->matrix[7][0] = 0;
    obj->matrix[7][1] = 1;
    obj->matrix[7][2] = 0;



 cout<<"\n object
==========================================================
==========\n";
    obj->print_obj();



// scaling object by multiplying
my_matrix *scaling_mat =  new my_matrix(4,0);
create_scaling_matrix(100,100,0,scaling_mat);
my_matrix *obj2 =  new my_matrix(r,0);
matrix_multiplication(obj, scaling_mat , obj2);


//=================================================
==shearing==============================


my_matrix *scaled_obj =  new my_matrix(r,0);
matrix_multiplication(result_after_projection , scaling_mat,  scaled_obj );







my_matrix *perspective_projection_matrix = new my_matrix(4,0);
create_perspective_matrix(60,2.5,perspective_projection_matrix);


cout<<"\nperspective matrix() :"<<endl;
perspective_projection_matrix->print_obj();


my_matrix *result_after_projection =  new my_matrix(r,0);


matrix_multiplication(result_after_rotation ,
perspective_projection_matrix , result_after_projection);


making_last_column_one(result_after_projection,r);




cout<<"result after projection\n";
result_after_projection->print_obj();


     int gd = DETECT, gm;
    char pathtodriver[] = "";
    initgraph(&gd, &gm, pathtodriver);




  //  draw_object(result2,GREEN);
  //   delay(2000);
```

```
draw_object(scaled_obj ,BLUE);



getch();
closegraph();


return 0;
}


//============== translation==================


// 7) Write a program to apply various 3D transformations on a 3D object
and then apply parallel
// and perspective projection on it.
// Write a program to apply various 3D transformations on a 3D object
and then apply parallel
// and perspective projection on it.



//  2 point perspective view
//  rotating 1 point perspective gives --> 2 point perspective



#include <iostream>
#include <conio.h>
#include <math.h>


#include "C:\Users\akash\OneDrive\Desktop\26th_april\dda.cpp"


#include <vector>
#include <utility>


// #include <graphics.h>
using namespace std;


class my_matrix{
   public:
      double** matrix ;
      int row;
      int col;


   my_matrix(){


   }


   my_matrix(int r,int v){


      row = r;
```

```
        col = 4; // for homogeneous


        matrix = new double*[r];


         // Allocate memory for each row
           for (int i = 0; i < r; ++i) {
              matrix[i] = new double[col]; // Create an array of integers for
each row
           }


            // Initialize each element to 1
           for (int i = 0; i < r; ++i) {
              for (int j = 0; j < col; ++j) {
                 matrix[i][j] = v;
              }
           }
    }


    void print_obj(){


  cout<<"--------------------------------------OBJECT-------------------------
------------------\n";
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
           std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
    cout<<"--------------------------------------------------------------------
------------------\n";
    }


    void deallocate_matrix(){
         // Free dynamically allocated memory
    for (int i = 0; i < row; ++i) {
       delete[] matrix[i];
    }
    delete[] matrix;
    }
};




void create_scaling_matrix(int sx,int sy,int sz, my_matrix *obj){
    obj->matrix[0][0] = sx;
    obj->matrix[1][1] = sy;
    obj->matrix[2][2] = sz;
    obj->matrix[3][3] = 1;


}


void create_translation_matrix(int tx,int ty,int tz, my_matrix *obj){
    obj->matrix[0][0] = 1;
    obj->matrix[1][1] = 1;
    obj->matrix[2][2] = 1;
    obj->matrix[3][3] = 1;


    obj->matrix[3][0] = tx;
    obj->matrix[3][1] = ty;




}
```

```
void create_perspective_matrix(double p,double yc,double zc,
my_matrix *obj){
    // obj->matrix[0][3] = p;
    // obj->matrix[1][3] = -1*(1/yc);
    // obj->matrix[2][3] = -1*(1/zc);
    // obj->matrix[3][3] = 1;
    // obj->matrix[2][2] = 0;


    // obj->matrix[0][0] = 1;
    // obj->matrix[1][1] = 1;

    obj->matrix[0][0] = 0.5;
    obj->matrix[1][1] = 1;
    obj->matrix[2][2] = 0;
    obj->matrix[3][3] = 1;


    obj->matrix[2][0] = 0.866;
    obj->matrix[3][1] = 2;
    obj->matrix[3][0] = 0 ;


    obj->matrix[0][3] = 0.346 ;
    obj->matrix[2][3] = -0.2 ;


}




void matrix_multiplication(my_matrix *obj , my_matrix *scaling ,
my_matrix *result){
     // Function to perform matrix multiplication



       if (obj->col != scaling->row) {
           std::cerr << "Error: Matrix dimensions mismatch for
multiplication." << std::endl;
           exit(1);
       }
       for (int i = 0; i < obj->row; ++i) {
           for (int j = 0; j < scaling ->col; ++j) {
               result -> matrix[i][j] = 0;



               for (int k = 0; k < obj -> col ; ++k) {
                   result -> matrix[i][j] +=  obj->matrix[i][k] * scaling -> matrix[k]
[j];
               }



           }
       }

}









void draw_object(my_matrix *obj,int colorr){   // only for 6 faces
    int x1=0;
    int y1=0;
```

```
    int x2=0;
    int y2=0;
    x1 = obj->matrix[0][0];
    y1 = obj->matrix[0][1];


    vector< pair< double, double> > points;
    vector< pair< double, double> > points2;


    for(int i=0;i< 4;i++){
        points.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));
        if(i==3){
            points.push_back(make_pair(obj->matrix[0][0],obj->matrix[0][1]));
        }
    }


     for(int i=4;i< 8;i++){
        points2.push_back(make_pair(obj->matrix[i][0],obj->matrix[i][1]));


    }


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


        //print  LINE 1
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points[i+1].first<< " , "<<points[i+1].second<<"   : : ";
        delay(1000);
        dda(points[i].first,   points[i].second    , points[i+1].first,
points[i+1].second , colorr );


        cout<<endl;
        cout<<points[i].first<< " , "<<points[i].second<<"   : : "
<<points2[i].first<< " , "<<points2[i].second<<"   : : ";


         delay(1000);
        dda(points[i].first , points[i].second,  points2[i].first ,
points2[i].second, colorr  );


        cout<<"\n----------------------------------------------------------"<<endl;
    }




   points2.push_back(make_pair(points2[0].first,points2[0].second));


    for(int i=0;i<points2.size()-1;i++){
        cout<<points2[i].first<< " , "<<points2[i].second<<"   : : "
<<points2[i+1].first<< " , "<<round(points2[i+1].second)<<"   : : \n";
        dda( points2[i].first  ,points2[i].second    , round( points2[i+1].first )  ,
round(points2[i+1].second) , colorr);
         delay(1000);
    }
}
```

```
inline double clamp(double x, double epsilon = 1e-6) {
    return (x < epsilon && x > -epsilon) ? 0.0 : x;
}
```

```
void making_last_column_one(my_matrix *obj,int row){


    for(int i =0; i<row ;i++ ){
        obj->matrix[i][0] =  clamp( (obj->matrix[i][0] / obj->matrix[i][3]));
        obj->matrix[i][1] =  clamp( (obj->matrix[i][1] / obj->matrix[i][3]) );
        obj->matrix[i][2] = clamp(  (obj->matrix[i][2] / obj->matrix[i][3])  );
        obj->matrix[i][3] =  clamp( (obj->matrix[i][3] / obj->matrix[i][3]) );
    }
}



int main(){


    int r = 8 ;
    my_matrix *obj;


if(r>=1){
    cout<<"matrix made"<<endl;
    obj  = new my_matrix(r,1);
    obj->print_obj();
}else{
    cout<<"invalid input";
    return 0;
}



    obj->matrix[0][0] = 0;
    obj->matrix[0][1] = 0;
    obj->matrix[0][2] = 1;


    obj->matrix[1][0] = 1;
    obj->matrix[1][1] = 0;
    obj->matrix[1][2] = 1;


    obj->matrix[2][0] = 1;
    obj->matrix[2][1] = 1;
    obj->matrix[2][2] = 1;


    obj->matrix[3][0] = 0;
    obj->matrix[3][1] = 1;
    obj->matrix[3][2] = 1;


    obj->matrix[4][0] = 0;
    obj->matrix[4][1] = 0;
    obj->matrix[4][2] = 0;


    obj->matrix[5][0] = 1;
    obj->matrix[5][1] = 0;
    obj->matrix[5][2] = 0;


    obj->matrix[6][0] = 1;
    obj->matrix[6][1] = 1;
    obj->matrix[6][2] = 0;


    obj->matrix[7][0] = 0;
    obj->matrix[7][1] = 1;
    obj->matrix[7][2] = 0;
```

```
int tx = 200 , sx2 = 50;
int ty = 0 , sy2 =50;
int tz = 0 ,   sz2 = 0;




my_matrix *translation_matrix = new my_matrix(4,0);
create_translation_matrix(tx,ty,tz,translation_matrix);


my_matrix *scaling_matrix2 = new my_matrix(4,0);;
create_scaling_matrix(sx2,sy2,sz2,scaling_matrix2);


my_matrix *perspective_projection_matrix = new my_matrix(4,0);
create_perspective_matrix(0,1000,1000
,perspective_projection_matrix);







cout<<"\nperspective matrix() :"<<endl;
perspective_projection_matrix->print_obj();


my_matrix *result_after_projection =  new my_matrix(r,0);
matrix_multiplication(obj, perspective_projection_matrix ,
result_after_projection);


making_last_column_one(result_after_projection,r);


    int gd = DETECT, gm;
   char pathtodriver[] = "";
   initgraph(&gd, &gm, pathtodriver);




my_matrix *result2 =  new my_matrix(r,0);  // 8  x 4     4 x 4 -->  8 x 4
matrix_multiplication(result_after_projection ,scaling_matrix2,result2);


    draw_object(result2,GREEN);


    delay(2000);


// TRANSLATION


my_matrix *result =  new my_matrix(r,0);  // 8  x 4     4 x 4 -->  8 x 4
matrix_multiplication(result2,translation_matrix ,result);

 draw_object(result,BLUE);

getch();
closegraph();


return 0;
}
```