**DEERWALK INSTITUTE OF TECHNOLOGY**

**Tribhuvan University**

**Faculties of Computer Science**

**A logo of a sea creature

Description automatically generated**

**Bachelors of Science in Computer Science and Information Technology (BSc. CSIT)**

**Course: Computer Graphics (CSC214)**

**Year/Semester: II/III**

**A Lab report on:**

**Implementation of Hidden Surface Removal**

Submitted by:

Name: Parth Poudyal

Roll: 1317

Submitted to:

Binod Sitaula

Department of Computer Science

**Theory**

Visible surface Detection

The process of hidden surface elimination involves identifying the visible parts of objects from a viewport, which requires significant processing time and may only be suitable for special types of objects rather than general geometric shapes. Choosing an appropriate visible surface detection method for a particular application depends on factors such as the complexity of the scene, the types of objects involved (considering texture, curvature, etc.), the available equipment, and whether the scene is static or animated. Visible surface algorithms are used in computer graphics to determine which surfaces of objects are visible when rendering. These algorithms can be categorized into two main approaches: object-space methods and image-space methods.

Object-space methods compare 3D objects within a scene before projection onto the screen, focusing on entire surfaces rather than individual pixels. They rely on coherence (regularity) to identify visible surfaces and detect overlapping surfaces before projection, making them resolution independent.

Image-space methods operate on a pixel-by-pixel basis, determining visibility by comparing pixel positions on the projection plane. Unlike object-space methods, image-space techniques are highly dependent on viewport resolution and are generally easier to implement since they focus on analyzing each pixel to establish visibility. This bottom-up approach is more commonly used in most visible surface detection methods. To improve performance, surfaces are often sorted in ascending or descending order to easily determine their depths relative to the viewport.

Z buffer method:

An Image space approach in which comparison of depths of each pixel from each of the surfaces on the projection plane is performed and stored in the Z buffer to determine the visible surfaces among the set of surfaces.

There is the need for two buffer areas

* Depth buffer:
  + stores the (z) depth values for the corresponding (X, Y) values.
  + Since the depths of the pixels may vary to extreme values, it is normalized for the values between 0 and 1 where 0 refers to the back of the clipping plane, while 1 refers to the front of the clipping plane.
* Refresh Buffer:
  + Stores the intensity of the (x, y) position
  + The visible surfaces have risen intensity raised, while others have the default intensity of the background.

**Algorithm**

* Initially all the depth values of the depth buffer are set to 0, while the values in the refresh buffer are set to the intensity of the background.
* Each pixel of the surfaces is processed as a scan line; serially missing out on no pixel.
  + for each pixel: Depth value is noted, and the intensity of the pixel is noted.
  + To note: the scan line passes along the z-axis. i.e., multiple surface's same x, y values are compared.
* The calculated depth value is compared to the values that were previously stored in the depth buffer
  + if the new values are lesser than the ones calculated at the z buffer, then the new value is discarded
  + if the values (new) are greater than the ones calculated at the z buffer is greater than the one stored, it is replaced with the value initially stored in the z buffer. correspondingly the intensity values are also stored in the refresh buffer.

z values come off the equation of the plane:

z =

Advantages:

* Since it is an image space method, it is easier to implement.
* It reduces the speed problem
* It processes one object at a time

Disadvantages:

* the use of depth and refresh buffer increases memory
* It is time consuming since it does not use the concept of a buffer

**Program**

#include <graphics.h>

#include <iostream>

#include <algorithm>

#include <vector>

#include <limits>

using namespace std;

const int WIDTH = 640;

const int HEIGHT = 480;

const int MAX\_DEPTH = 1000;

float zBuffer[WIDTH][HEIGHT]; // Z-buffer array

void initializeZBuffer() {

    for(int i = 0; i < WIDTH; i++) {

        for(int j = 0; j < HEIGHT; j++) {

            zBuffer[i][j] = MAX\_DEPTH; // Initialize all pixels to maximum depth

        }

    }

}

void drawTriangle(int x1, int y1, float z1, int x2, int y2, float z2, int x3, int y3, float z3, int color) {

    int ymin = min({y1, y2, y3});

    int ymax = max({y1, y2, y3});

    for(int y = ymin; y <= ymax; y++) {

        vector<pair<int, float>> intersections;

        auto edgeIntersect = [&](int x0, int y0, float z0, int x1, int y1, float z1) {

            if (y0 == y1) return; // Horizontal line, skip

            if (y0 > y1) {

                swap(x0, x1);

                swap(y0, y1);

                swap(z0, z1);

            }

            if (y >= y0 && y <= y1) {

                float t = float(y - y0) / (y1 - y0);

                int x = x0 + t \* (x1 - x0);

                float z = z0 + t \* (z1 - z0);

                intersections.push\_back({x, z});

            }

        };

        edgeIntersect(x1, y1, z1, x2, y2, z2);

        edgeIntersect(x2, y2, z2, x3, y3, z3);

        edgeIntersect(x3, y3, z3, x1, y1, z1);

        if (intersections.size() >= 2) {

            sort(intersections.begin(), intersections.end());

            int xStart = intersections[0].first;

            int xEnd = intersections[1].first;

            float zStart = intersections[0].second;

            float zEnd = intersections[1].second;

            for (int x = xStart; x <= xEnd; x++) {

                if (x < 0 || x >= WIDTH || y < 0 || y >= HEIGHT) continue;

                float t = (float)(x - xStart) / (xEnd - xStart);

                float z = zStart + t \* (zEnd - zStart);

                if (z < zBuffer[x][y]) {

                    zBuffer[x][y] = z;

                    putpixel(x, y, color);

                }

            }

        }

    }

}

int main() {

    int gd = DETECT, gm;

    initgraph(&gd, &gm, (char\*)"");

    initializeZBuffer();

    drawTriangle(100, 100, 10, 300, 150, 50, 200, 300, 30, WHITE);

    drawTriangle(200, 150, 20, 400, 200, 70, 300, 350, 60, RED);

    drawTriangle(150, 250, 5, 350, 300, 40, 250, 400, 20, GREEN);

    getch();

    closegraph();

    return 0;

}

A group of triangles in different colors

AI-generated content may be incorrect.