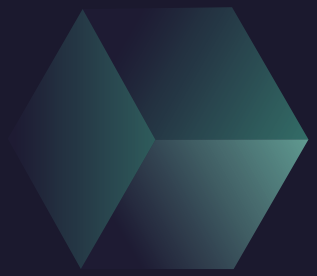
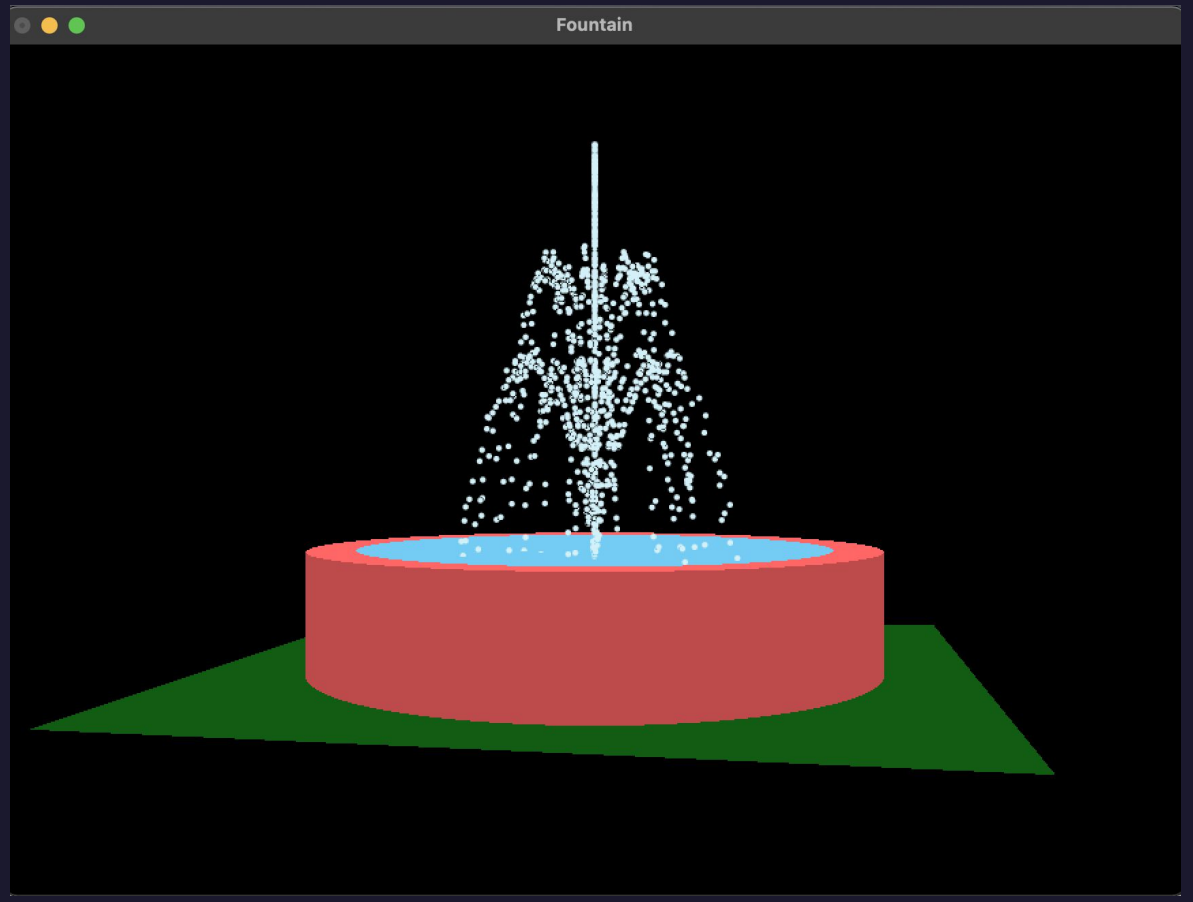


INTERACTIVE COMPUTER GRAPHICS

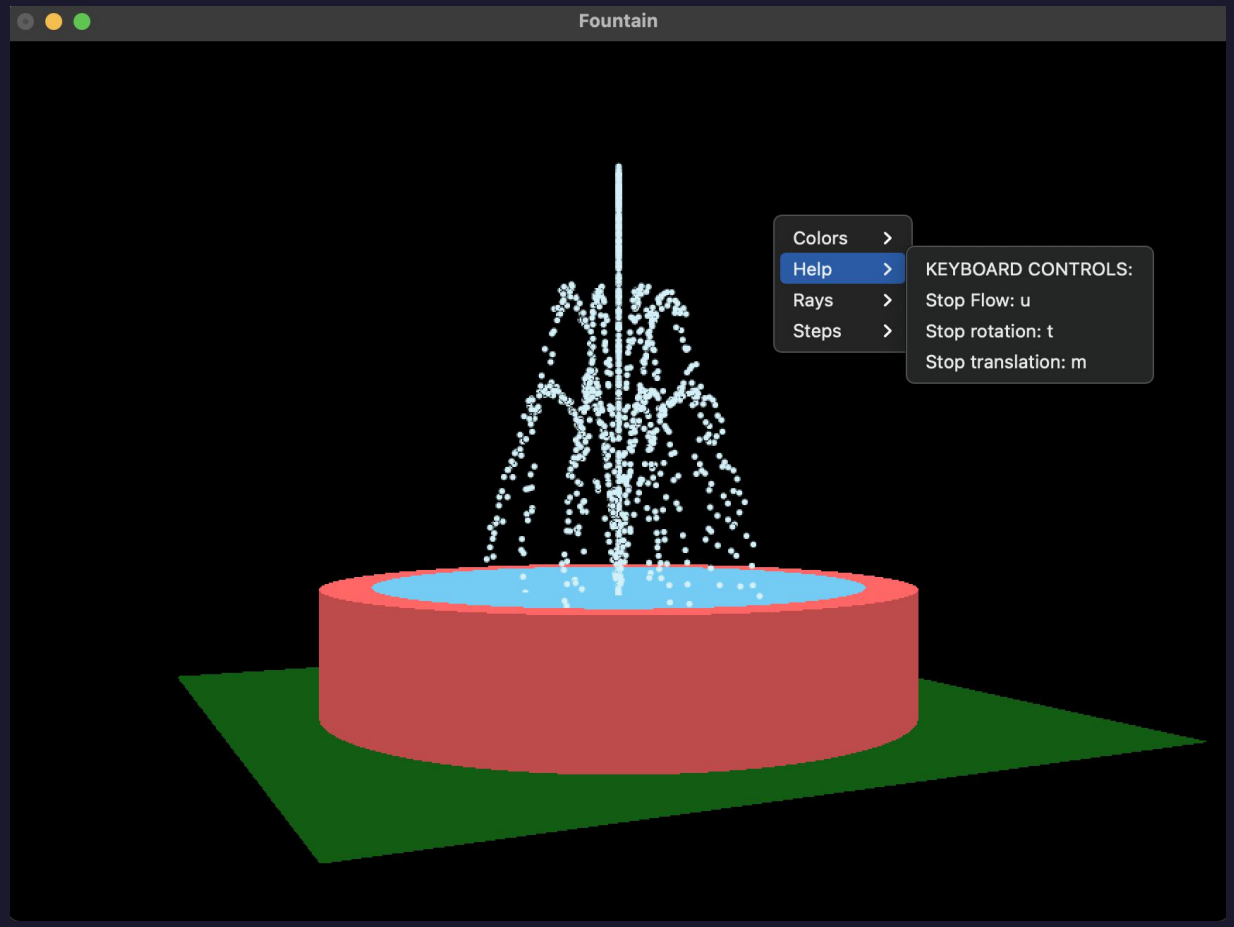
SEMINAR TOPIC : MOVING FOUNTAIN using OPENGL

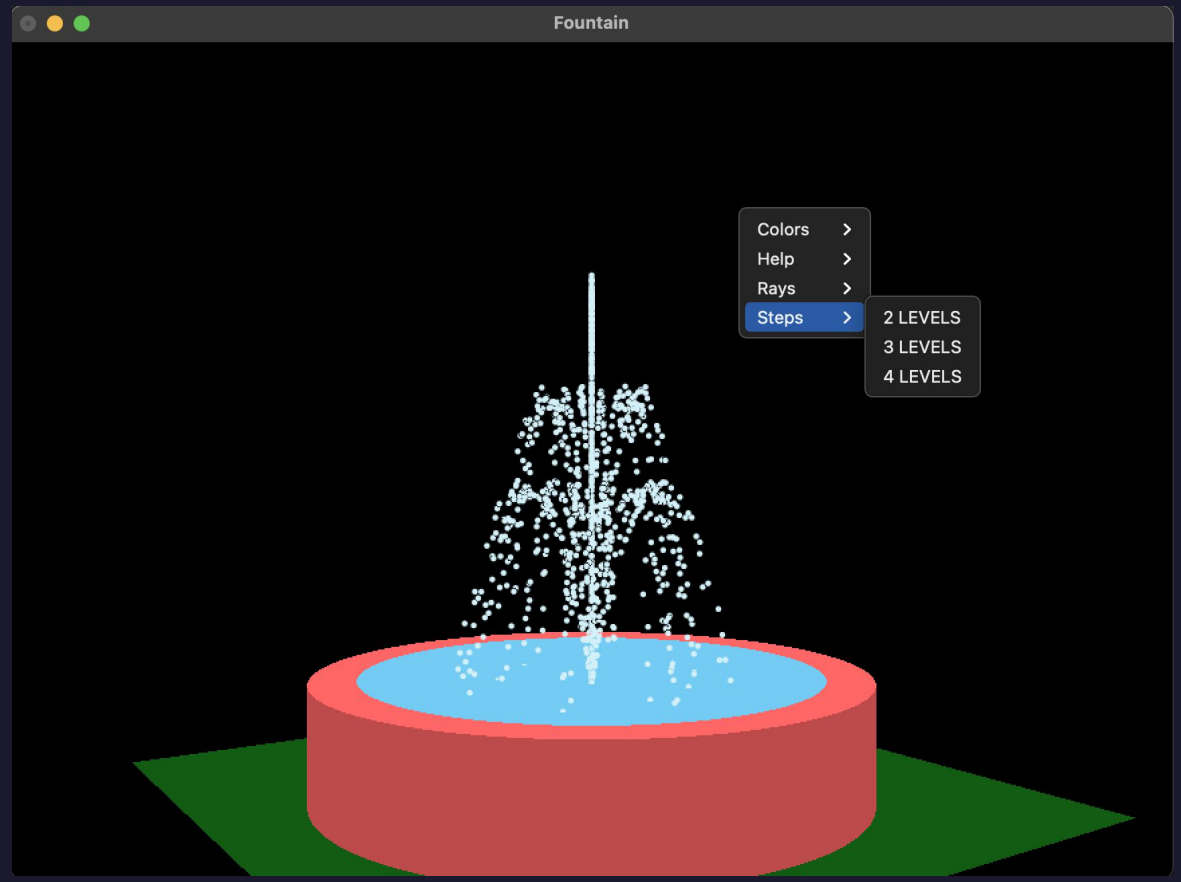
DARA SANJANA
CED18I015

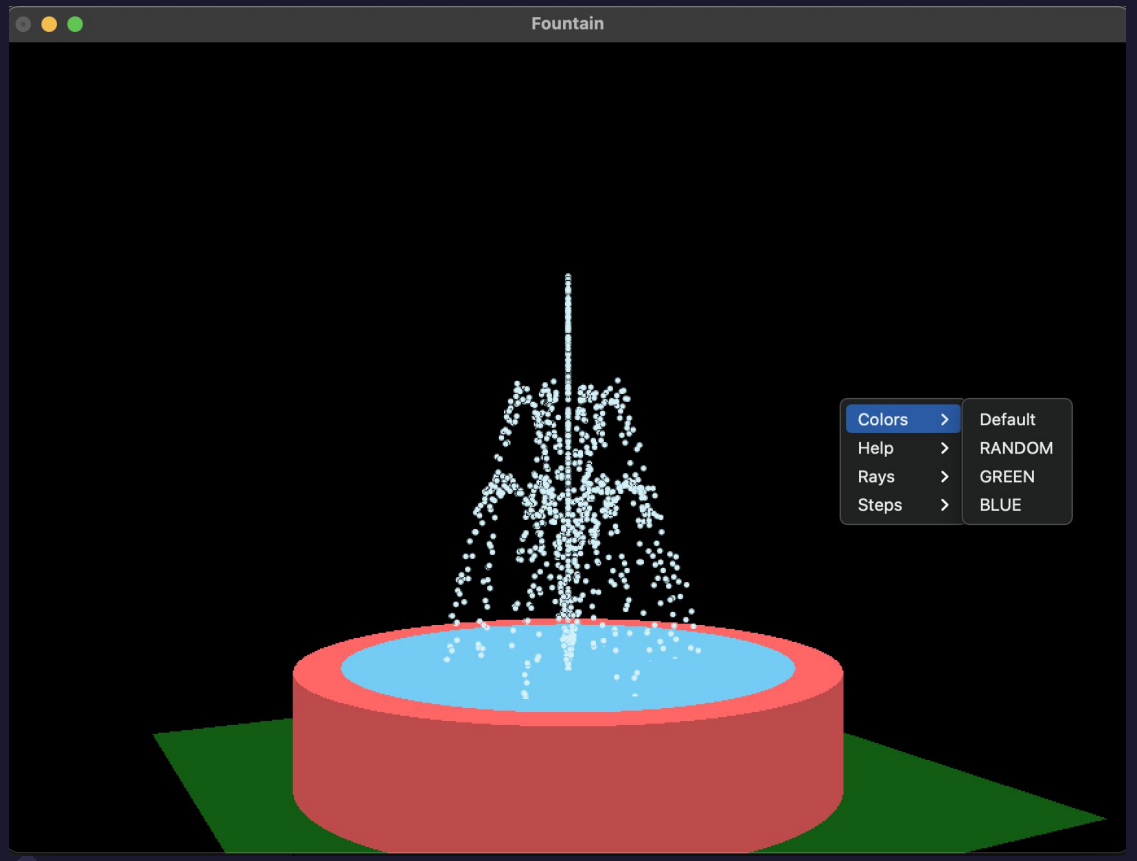
A flowing fountain that rotates and translates around y-axis.

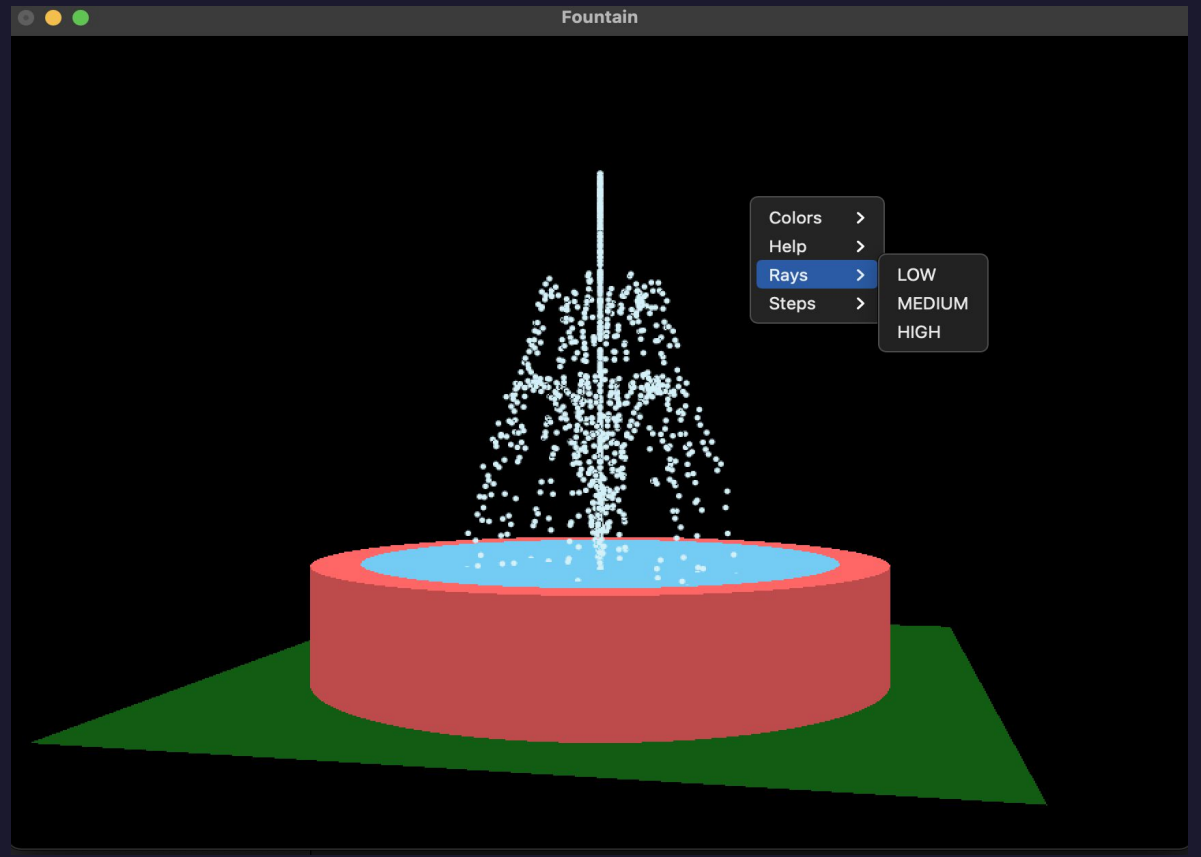


Operations that can be done: (Right click on the screen)






Fountain
Colors
Help
Rays
Steps
Default
RANDOM
GREEN
BLUE


Fountain
Colors
Help
Rays
Steps
LOW
MEDIUM
HIGH

CODE EXPLAINED :

void CreateList(**void**) : Calculates vertices for the base of the fountain, then draws the base with the calculated vertices
every time the screen redisplays.

```
glNewList(ListNum, GL_COMPILE);

    glBegin(GL_QUADS);
    //ground quad:
    glColor3ub(18, 92, 19);
    glVertex3f(-OuterRadius*1.3,0.0,OuterRadius*1.3);
    glVertex3f(-OuterRadius*1.3,0.0,-OuterRadius*1.3);
    glVertex3f(OuterRadius*1.3,0.0,-OuterRadius*1.3);
    glVertex3f(OuterRadius*1.3,0.0,OuterRadius*1.3);
    //stone:
    for (int j = 1; j < 3; j++)
    {
        if (j == 1) glColor3ub(255, 103, 103);
        if (j == 2) glColor3ub(189, 75, 75);
        for (i = 0; i<NumOfVerticesStone-1; i++)
        {
            glVertex3fv(&Vertices[i+NumOfVerticesStone*j].x);
            glVertex3fv(&Vertices[i].x);
            glVertex3fv(&Vertices[i+1].x);
            glVertex3fv(&Vertices[i+NumOfVerticesStone*j+1].x);
        }
        glVertex3fv(&Vertices[i+NumOfVerticesStone*j].x);
        glVertex3fv(&Vertices[i].x);
        glVertex3fv(&Vertices[0].x);
        glVertex3fv(&Vertices[NumOfVerticesStone*j].x);
    }
```

void KeyDown(**unsigned char** key, **int** x, **int** y) : Calls functions that are chosen by keyboard clicks.

In main function, menu is created and f
unctions are called according to the mouse actions.

```
void KeyDown(unsigned char key, int x, int y)
{
    switch(key)
    {
    case 27:    //ESC
        exit(0);
        break;
    case 't':
        DoTurn = !DoTurn;
        break;
    case 'm':
        DoMoveUp = !DoMoveUp;
        break;
    case 'u':
        DoUpdateScene = !DoUpdateScene;
        break;
    }
}
```

void InitFountain(**void**) : Calculates initial speeds , times needed, acceleration factor for each fountain drop. These factors are given on the basis of ray index, step index and also some pre determined constants taken from calculations from physics.

```
void InitFountain(void)
{
    //This function needn't be and isn't speed optimized
    FountainDrops = new CDrop [ DropsComplete ];
    FountainVertices = new SVertex [ DropsComplete ];
    SVertex NewSpeed;
    GLfloat DropAccFactor; //different from AccFactor because of the random change
    GLfloat TimeNeeded;
    GLfloat StepAngle; //Angle, which the ray gets out of the fountain with
    GLfloat RayAngle;    //Angle you see when you look down on the fountain
    GLint i,j,k;
    for (k = 0; k <Steps; k++)
    {
        for (j = 0; j < RaysPerStep; j++)
        {
            for (i = 0; i < DropsPerRay; i++)
            {
                DropAccFactor = AccFactor + GetRandomFloat(0.0005);
                StepAngle = AngleOfDeepestStep + (90.0-AngleOfDeepestStep)* GLfloat(k) / (Steps-1);
                //This is the speed caused by the step:
                NewSpeed.x = cos ( StepAngle * PI / 180.0) * (0.2+0.04*k);
                NewSpeed.y = sin ( StepAngle * PI / 180.0) * (0.2+0.04*k);
                //This is the speed caused by the ray:

                RayAngle = (GLfloat)j / (GLfloat)RaysPerStep * 360.0;
                //for the next computations "NewSpeed.x" is the radius. Care! Dont swap the two
                //lines, because the second one changes NewSpeed.x!
                NewSpeed.z = NewSpeed.x * sin ( RayAngle * PI /180.0);
                NewSpeed.x = NewSpeed.x * cos ( RayAngle * PI /180.0);

                //Calculate how many steps are required, that a drop comes out and falls down again
                TimeNeeded = NewSpeed.y/ DropAccFactor;
                FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetConstantSpeed ( NewSpeed );
                FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetAccFactor (DropAccFactor);
                FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetTime(TimeNeeded * i / DropsPerRay);
            }
        }
    }
```

void DrawFountain(**void**) : Every time the screen redisplays, fountain drops are updated with new position

```
void DrawFountain(void)
{
    glColor4f(0.8,0.8,0.8,0.8);
    if (DoUpdateScene)
    for (int i = 0; i < DropsComplete; i++)
    {
        FountainDrops[i].GetNewPosition(&FountainVertices[i]);
    }
    glColor3ub(212,241,249);
    if(flag==1)
        glColor3ub(X[0],X[1],X[2]);
    else if(flag==2)
        glColor3ub(0,255,0);
    else if(flag==3)
        glColor3ub(0,0,255);
    glDrawArrays(GL_POINTS,0,DropsComplete);
}
```

Fountain Physics:

```
CDrop * FountainDrops;
SVertex * FountainVertices;
GLint Steps = 3;   //a fountain has several steps, each wit
GLint RaysPerStep = 8;
GLint DropsPerRay = 50;
GLfloat DropsComplete = Steps * RaysPerStep * DropsPerRay;
GLfloat AngleOfDeepestStep = 80;
GLfloat AccFactor = 0.011;
```

```
for (k = 0; k <Steps; k++)
{
    for (j = 0; j < RaysPerStep; j++)
    {
        for (i = 0; i < DropsPerRay; i++)
        {
            DropAccFactor = AccFactor + GetRandomFloat(0.0005);
            StepAngle = AngleOfDeepestStep + (90.0-AngleOfDeepestStep)* GLfloat(k) / (Steps-1);
            //This is the speed caused by the step:
            NewSpeed.x = cos ( StepAngle * PI / 180.0) * (0.2+0.04*k);
            NewSpeed.y = sin ( StepAngle * PI / 180.0) * (0.2+0.04*k);
            //This is the speed caused by the ray:

            RayAngle = (GLfloat)j / (GLfloat)RaysPerStep * 360.0;
            //for the next computations "NewSpeed.x" is the radius. Care! Dont swap the two
            //lines, because the second one changes NewSpeed.x!
            NewSpeed.z = NewSpeed.x * sin ( RayAngle * PI /180.0);
            NewSpeed.x = NewSpeed.x * cos ( RayAngle * PI /180.0);

            //Calculate how many steps are required, that a drop comes out and falls down again
            TimeNeeded = NewSpeed.y/ DropAccFactor;
            FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetConstantSpeed ( NewSpeed );
            FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetAccFactor (DropAccFactor);
            FountainDrops[i+j*DropsPerRay+k*DropsPerRay*RaysPerStep].SetTime(TimeNeeded * i / DropsPerRay);
        }
    }
}
```

```
void CDrop::SetConstantSpeed(SVertex NewSpeed)
{
    ConstantSpeed = NewSpeed;
}

void CDrop::SetAccFactor (GLfloat NewAccFactor)
{
    AccFactor = NewAccFactor;
}

void CDrop::SetTime(GLfloat NewTime)
{
    time = NewTime;
}

void CDrop::GetNewPosition(SVertex * PositionVertex)
{
    SVertex Position;
    time += 1;
    Position.x = ConstantSpeed.x * time;
    Position.y = ConstantSpeed.y * time - AccFactor * time * time;
    Position.z = ConstantSpeed.z * time;
    PositionVertex->x = Position.x;
    PositionVertex->y = Position.y + WaterHeight;
    PositionVertex->z = Position.z;
    if (Position.y < 0.0)
    {
        time = time - int(time);
        if (time > 0.0)
            time -= 1.0;
    }

}
```

GLfunctions used:

glutReshapeFunc(Reshape); The reshape function **defines what to do when the window is resized**. It must have a void return type, and takes two int parameters (the new width and height of the window).

glutKeyboardFunc(KeyDown) : The new keyboard callback function. glutKeyboardFunc **sets the keyboard callback for the current window**. When a user types into the window, each key press generating an ASCII character will generate a keyboard callback.

glutCreateMenu(CMain); glutCreateMenu creates a new **pop-up menu** and returns a unique small integer identifier. The range of allocated identifiers starts at one. The menu identifier range is separate from the window identifier range. Implicitly, the current menu is set to the newly created menu.

glutIdleFunc(Display); glutIdleFunc sets the **global** idle callback to be func so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. If enabled, the idle callback is continuously called when events are not being received. The callback routine has no parameters.

glutPostRedisplay() : glutPostRedisplay **marks the current window as needing to be redisplayed**. Mark the normal plane of current window as needing to be redisplayed. The next iteration through glutMainLoop, the window's display callback will be called to redisplay the window's normal plane.

glTranslatef : The glTranslatef function produces the translation specified by (x, y, z). The translation vector is used to compute a 4x4 translation matrix: The current matrix (see glMatrixMode) is multiplied by this translation matrix, with the product replacing the current matrix.

glRotatef : The glRotatef function computes a matrix that performs a counterclockwise rotation of angle degrees about the vector from the origin through the point (x, y, z). The current matrix (see glMatrixMode) is multiplied by this rotation matrix, with the product replacing the current matrix.

glDrawArrays : glDrawArrays specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to glDrawArrays.

glNewList(ListNum, GL_COMPILE):

Display lists are groups of GL commands that have been stored for subsequent execution. Display lists are created with glNewList. All subsequent commands are placed in the display list, in the order issued, until glEndList is called.

glNewList has two arguments. The first argument, list, is a positive integer that becomes the unique name for the display list. Names can be created and reserved with glGenLists and tested for uniqueness with glIsList. The second argument, mode, is a symbolic constant that can assume one of two values:

THANK YOU