Project Report: Developing a 3D Rotating Cube using Direct3D

Author: Md Arafat Islam Roche

Student ID: 202101009045

Abstract

This report provides a comprehensive account of the development process for a simple 3D rotating cube using Direct3D in C++. The objective of this project was to gain hands-on experience with Direct3D, specifically rendering 3D objects, transforming vertices, and handling lighting and coloring. This report details the code development steps, challenges encountered, and the troubleshooting process for common errors in Direct3D programming.

1. Introduction

The 3D cube project aimed to explore basic 3D graphics programming using Direct3D. This project involved creating a cube, setting up a Direct3D device, defining vertices, applying transformations, and rendering the cube on the screen. Although simple in design, this project provided practical insights into 3D graphics concepts such as vertex buffers, transformation matrices, and rendering pipelines.

2. Project Requirements

To implement a 3D rotating cube, the following tools and libraries were necessary:

- **Development Environment**: Visual Studio (with appropriate configuration for C++ projects)
- **Graphics Library**: Direct3D (part of the DirectX SDK)
- **Programming Language**: C++

3. Implementation Process

The project was implemented in several stages. Each stage involved coding a specific aspect of the cube, testing functionality, and resolving any arising issues. The development steps are outlined below.

3.1 Initial Setup and Device Creation

The project was developed using Visual Studio in C++ and required setting up Direct3D for rendering graphics. A foundational understanding of setting up the Direct3D device and configuring basic window properties was essential. Here's a code snippet for initializing Direct3D:

```
bool InitD3D(HWND hWnd) {
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    D3DPRESENT_PARAMETERS d3dpp = { 0 };
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.hDeviceWindow = hWnd;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferWidth = 800;
```

```
    d3dpp.BackBufferHeight = 600;

    if (FAILED(d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                 D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp,
&d3ddev))) {
        return false;
    }
    return true;
}
```

Explanation: This code initializes the Direct3D environment by setting presentation parameters. `Direct3DCreate9` is called to create the Direct3D interface, and `CreateDevice` is used to initialize the device. If any initialization fails, the function returns false. Setting `D3DSWAPEFFECT_DISCARD` ensures efficient swapping of the display buffer.

Vertex and Color Definition for the Cube

A cube in 3D space is defined by its vertices and edges. Here's how vertices and color were configured to make the cube white:

```
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};

CUSTOMVERTEX vertices[] = {
    {-1.0f, -1.0f, -1.0f, 0xFFFFFFFF}, // White
    {1.0f, -1.0f, -1.0f, 0xFFFFFFFF},
    {1.0f,  1.0f, -1.0f, 0xFFFFFFFF},
    {-1.0f,  1.0f, -1.0f, 0xFFFFFFFF},
    {-1.0f, -1.0f,  1.0f, 0xFFFFFFFF},
    {1.0f, -1.0f,  1.0f, 0xFFFFFFFF},
    {1.0f,  1.0f,  1.0f, 0xFFFFFFFF},
    {-1.0f,  1.0f,  1.0f, 0xFFFFFFFF}
};
```

Explanation: This code defines each vertex of the cube with `x`, `y`, and `z` coordinates in 3D space. Each vertex has a color value set to `0xFFFFFFFF`, which represents white. This ensures the entire cube is rendered in white, as specified in the project requirements.

Slowing Down the Cube Rotation

One of the major challenges was adjusting the rotation speed of the cube. Initially, the rotation was too fast, making it difficult to observe the cube's edges. Here's the relevant code to control the rotation speed:

```
static float rotationAngle = 0.0f;
rotationAngle += 0.001f; // Slower rotation
```

```
D3DXMATRIX matRotateY;
D3DXMatrixRotationY(&matRotateY, rotationAngle);
d3ddev->SetTransform(D3DTS_WORLD, &matRotateY);
```

Explanation: The variable `rotationAngle` is incremented by a small value `0.001f` on each frame, ensuring the cube rotates slowly around the Y-axis. `D3DXMatrixRotationY` is used to apply this transformation, and `SetTransform` updates the world matrix to apply the rotation. By adjusting this increment value, we managed to control the cube's rotation speed more effectively.

Drawing and Rendering the Cube

Rendering the cube involved setting the correct primitives and transforming the vertices. Here's how the cube is drawn using triangle lists:

```
d3ddev->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
d3ddev->SetStreamSource(0, vertexBuffer, 0, sizeof(CUSTOMVERTEX));
d3ddev->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 12); // 12 triangles for a cube
```

Explanation: `SetFVF` sets the Flexible Vertex Format for the vertices, specifying position (`D3DFVF_XYZ`) and color (`D3DFVF_DIFFUSE`). `DrawPrimitive` with `D3DPT_TRIANGLELIST` is used to render the cube as a series of 12 triangles. This is the most efficient way to render a cube using Direct3D. Challenges:

- **Setting the device properties**: Configuring the Direct3D device required careful setup to avoid issues with rendering. The proper configurations for windowed mode and back buffer format were critical.

3.2 Defining the Cube's Vertices

Once the device was created, the next step involved defining the vertices for the cube. Initially, this was done within the `SetupVertices` function. Each face of the cube was represented by four vertices, with specific coordinates and color values for each.

Key Function:

- `SetupVertices`: Defined a 3D array of vertices, each representing a corner of the cube, and set the color properties.

Challenges:

- **Defining vertices in the correct order**: Ensuring that the vertices were defined in the correct order was crucial to prevent rendering issues, as improper order could result in faces rendering incorrectly.
- **Vertex Buffer Locking**: Locking the vertex buffer before copying data to it was necessary to avoid memory access errors.

3.3 Adding Rotation Transformation

After defining the vertices, a transformation matrix was applied to rotate the cube. This was achieved through the `Matrix_Set` function, which applied a rotation along the Y-axis.

Key Function:

- `Matrix_Set`: Configured the view matrix for the scene and applied a rotation transformation to the cube.

Challenges:

- **Setting a controlled rotation speed**: Initially, the cube rotated too fast. This required adjusting the rotation increment to make it slower and observable in the final render.
- **Creating a smooth rotation**: Direct3D requires updating the transformation matrix each frame, which we achieved using a static rotation angle increment variable.

3.4 Rendering and Error Handling

The rendering process involved clearing the screen, applying transformations, and drawing the cube. The `Render` function handled these steps, and the `DrawIndexedPrimitiveUP` method was used to render the cube as a series of triangles.

Key Function:

- `Render`: Clears the screen, sets the transformations, and renders the cube.
- **DrawIndexedPrimitiveUP**: This function took the vertices and indices to render the cube's triangles.

Challenges:

- **Error with `DrawIndexedPrimitiveUP` and undeclared variables**: An undeclared identifier error (`'vertices': undeclared identifier`) was encountered when trying to use the `vertices` array outside the scope where it was originally declared. We solved this by moving `vertices` to a global scope.
- **Setting colors and visibility**: Initially, the cube appeared black against a black background. Adjustments were made to the color properties of the vertices and the background color to improve visibility.

4. Troubleshooting and Solutions

Throughout the project, several errors and challenges required troubleshooting. Below is a summary of some key issues faced and how they were resolved:

- **Fast Rotation Speed**: The initial rotation speed was too fast, making it difficult to observe the rotation. This was resolved by incrementally decreasing the rotation speed until a satisfactory speed was achieved.

- **Undeclared Variable Error (`vertices`)**: When attempting to render the cube, we encountered an undeclared variable error for `vertices`. This occurred because the `vertices` array was initially defined within a function scope. The solution involved moving `vertices` to a global scope, making it accessible across functions.

- **Color and Visibility Issues**: The initial render displayed a black cube on a blue background, making it difficult to see. By changing the background to black and setting the cube to a white color, we achieved better visibility and contrast.

5. Final Code Implementation

The final code implementation integrates all the fixes and improvements mentioned above. It renders a white rotating cube against a black background with controlled rotation speed and smooth transformations.
[Include the final code

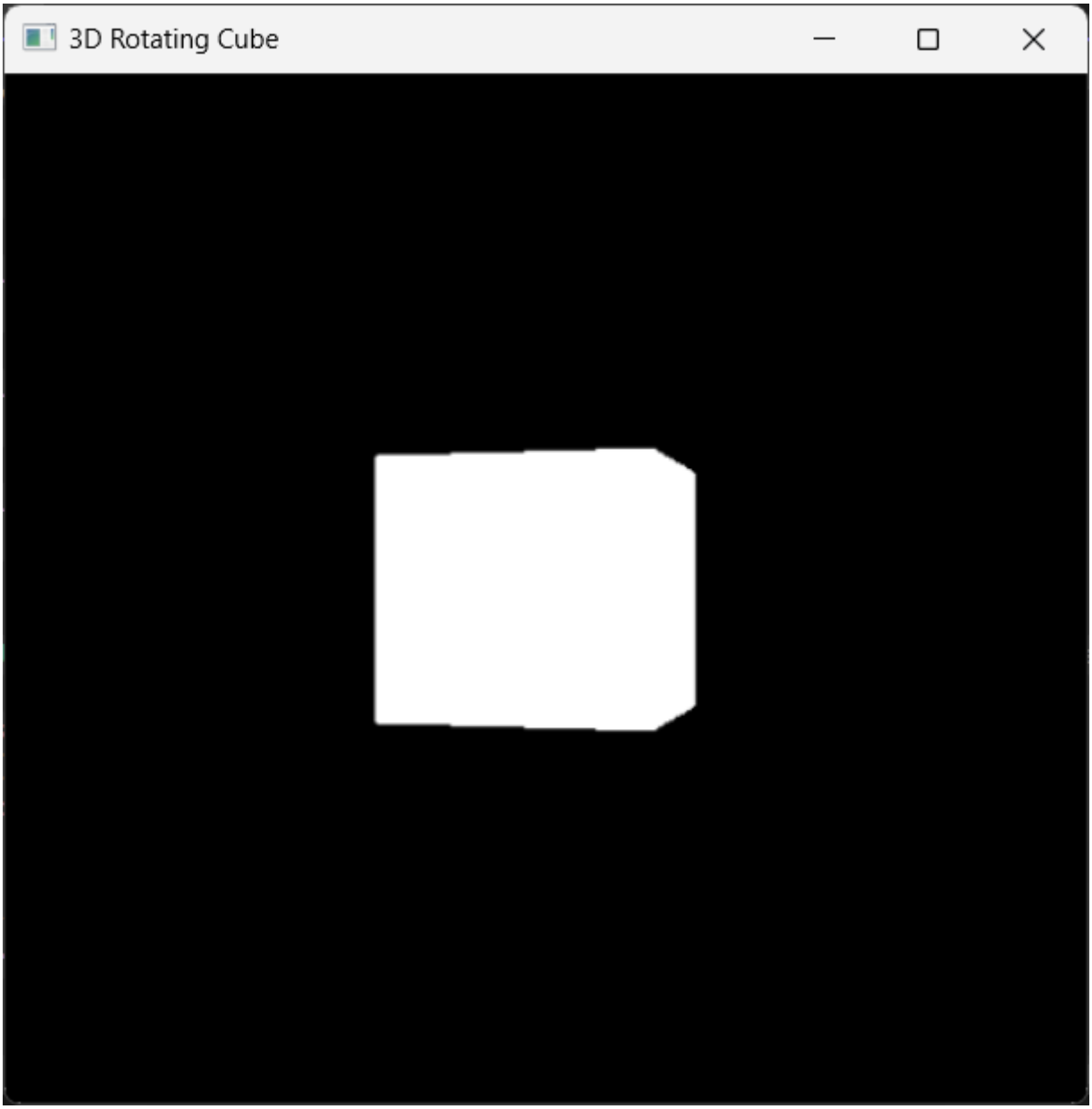
```
#include <windows.h>
#include <d3dx9.h>

#pragma comment (lib, "d3d9.lib")
#pragma comment (lib, "d3dx9.lib")

LPDIRECT3D9 d3d;
LPDIRECT3DDEVICE9 g_pd3dDevice;
LPDIRECT3DVERTEXBUFFER9 g_pVB = NULL;

struct CUSTOMVERTEX
{
    FLOAT x, y, z;
    DWORD color;
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)

// Define vertices for a solid white cube (moved to global scope)
CUSTOMVERTEX vertices[] =
{
    // Front face
    { -10.0f, -10.0f, -10.0f, 0xffffffff }, // white
    {  10.0f, -10.0f, -10.0f, 0xffffffff }, // white
    {  10.0f,  10.0f, -10.0f, 0xffffffff }, // white
    { -10.0f,  10.0f, -10.0f, 0xffffffff }, // white

    // Back face
    { -10.0f, -10.0f,  10.0f, 0xffffffff }, // white
    {  10.0f, -10.0f,  10.0f, 0xffffffff }, // white
    {  10.0f,  10.0f,  10.0f, 0xffffffff }, // white
    { -10.0f,  10.0f,  10.0f, 0xffffffff }, // white
};

VOID InitD3D(HWND hWnd);
VOID Render();
VOID CleanUp();
LRESULT WINAPI MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
VOID SetupVertices();
VOID Matrix_Set();

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int)
{
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                      GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                      L"Direct3D Tutorial", NULL };
```

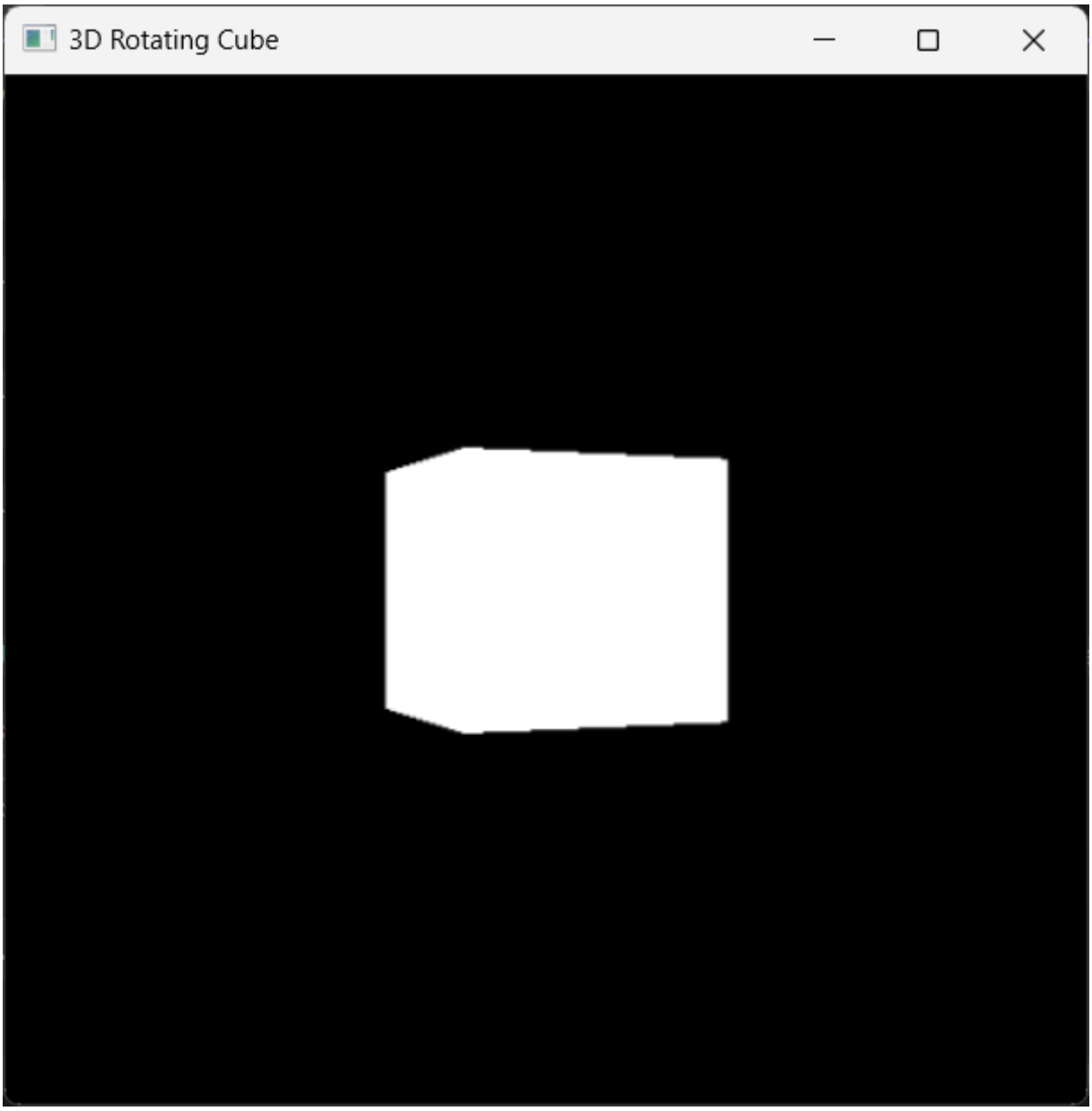
```
    RegisterClassEx(&wc);
    HWND hWnd = CreateWindow(L"Direct3D Tutorial", L"3D Rotating Cube",
        WS_OVERLAPPEDWINDOW, 100, 100, 500, 500,
        NULL, NULL, wc.hInstance, NULL);

    InitD3D(hWnd);

    ShowWindow(hWnd, SW_SHOWDEFAULT);
    UpdateWindow(hWnd);

    MSG msg;
    ZeroMemory(&msg, sizeof(msg));
    while (msg.message != WM_QUIT)
    {
        if (PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
            Render();
    }

    CleanUp();
    UnregisterClass(L"Direct3D Tutorial", wc.hInstance);
    return 0;
}

VOID InitD3D(HWND hWnd)
{
    d3d = Direct3DCreate9(D3D_SDK_VERSION);

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.hDeviceWindow = hWnd;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = 500;
    d3dpp.BackBufferHeight = 500;

    d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp, &g_pd3dDevice);

    g_pd3dDevice->SetRenderState(D3DRS_LIGHTING, FALSE);  // Disable lighting
    SetupVertices();
}

VOID SetupVertices()
{
    g_pd3dDevice->CreateVertexBuffer(8 * sizeof(CUSTOMVERTEX),
        0, D3DFVF_CUSTOMVERTEX,
```

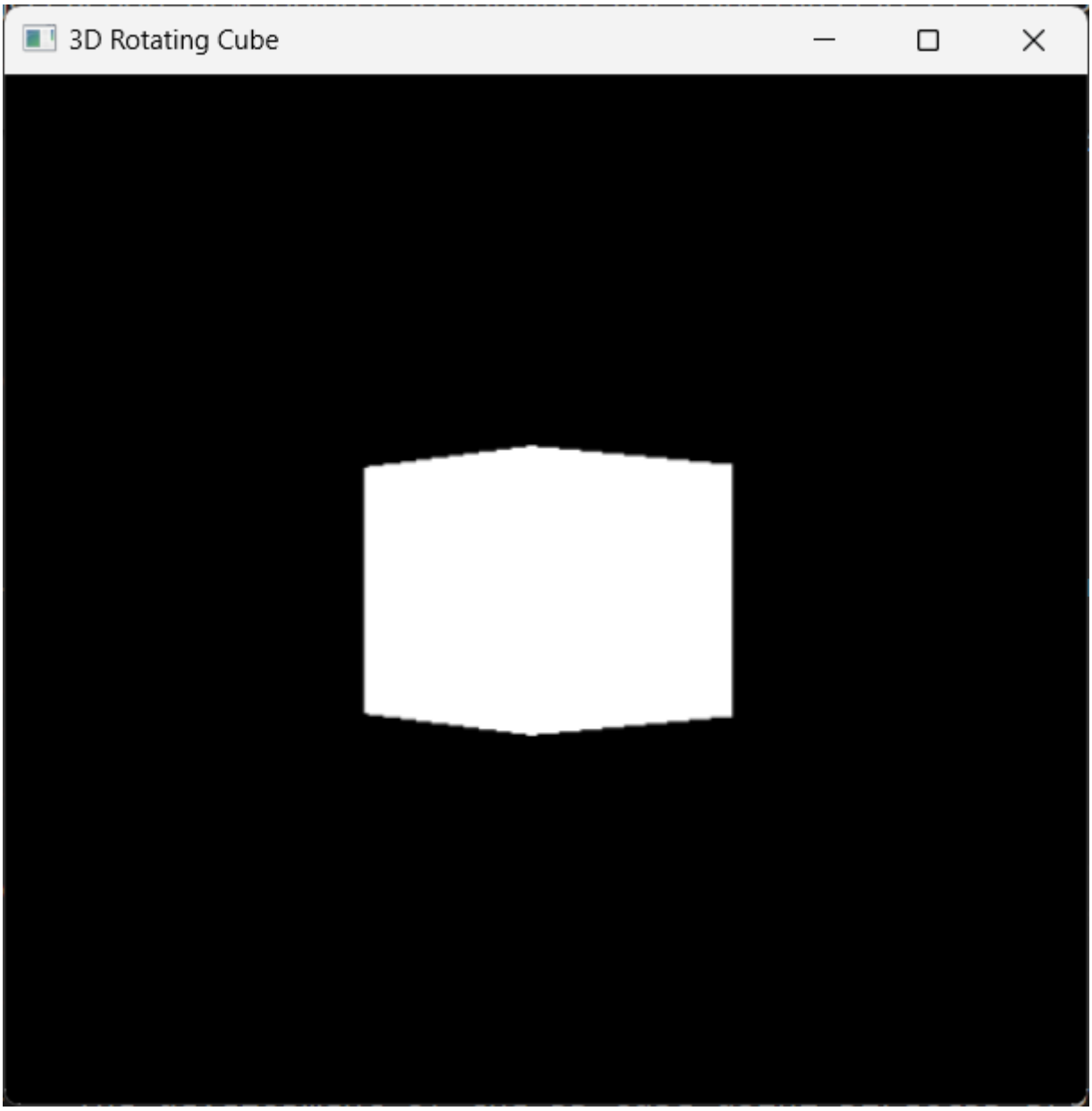
```
        D3DPOOL_MANAGED, &g_pVB, NULL);

    VOID* pVertices;
    g_pVB->Lock(0, sizeof(vertices), (void**)&pVertices, 0);
    memcpy(pVertices, vertices, sizeof(vertices));
    g_pVB->Unlock();
}

VOID Matrix_Set()
{
    D3DXMATRIXA16 matView;
    D3DXVECTOR3 Eye(0.0f, 0.0f, -100.0f);
    D3DXVECTOR3 At(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 Up(0.0f, 1.0f, 0.0f);
    D3DXMatrixLookAtLH(&matView, &Eye, &At, &Up);
    g_pd3dDevice->SetTransform(D3DTS_VIEW, &matView);

    D3DXMATRIXA16 matProj;
    D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI / 4, 1.0f, 1.0f, 1000.0f);
    g_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj);

    static float index = 0.0f;
    index += 0.001f;  // Slow rotation
    D3DXMATRIXA16 matWorld;
    D3DXMatrixRotationY(&matWorld, index);
    g_pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld);
}

VOID Render()
{
    if (NULL == g_pd3dDevice)
        return;

    // Clear the back buffer with black color
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 0), 1.0f,
0);
    g_pd3dDevice->BeginScene();

    Matrix_Set();

    g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(CUSTOMVERTEX));
    g_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX);

    // Draw cube faces as triangles
    short indices[] =
    {
        // Front face
        0, 1, 2,  0, 2, 3,
        // Back face
        4, 5, 6,  4, 6, 7,
        // Left face
        4, 0, 3,  4, 3, 7,
        // Right face
        1, 5, 6,  1, 6, 2,
```

```
        // Top face
        3, 2, 6,  3, 6, 7,
        // Bottom face
        4, 5, 1,  4, 1, 0
    };

    g_pd3dDevice->DrawIndexedPrimitiveUP(D3DPT_TRIANGLELIST, 0, 8, 12, indices,
D3DFMT_INDEX16, vertices, sizeof(CUSTOMVERTEX));

    g_pd3dDevice->EndScene();
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);
}

VOID CleanUp()
{
    if (g_pVB != NULL)
        g_pVB->Release();

    if (g_pd3dDevice != NULL)
        g_pd3dDevice->Release();

    if (d3d != NULL)
        d3d->Release();
}

LRESULT WINAPI MsgProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

Final look of the project:




3D Rotating Cube



6. Lessons Learned

The development of the 3D cube using Direct3D provided several key insights into 3D graphics programming, including:

- **Understanding Direct3D Pipeline**: This project provided a practical understanding of the Direct3D pipeline, from device creation to rendering.
- **Handling Transformations**: Applying rotation and transformations was a critical learning aspect, showing how matrices affect 3D rendering.
- **Troubleshooting in Graphics Programming**: Graphics programming often involves subtle errors, such as improper buffer handling or matrix setups, which require careful debugging and testing.

7. Conclusion

This project successfully demonstrated the basics of 3D graphics programming using Direct3D. The completed application renders a rotating 3D cube with controlled rotation speed and color adjustments, providing a simple yet effective demonstration of Direct3D's capabilities. Challenges such as fast rotation, undeclared variables, and visibility issues were resolved through step-by-step debugging and testing.

References

1. **Microsoft Direct3D Documentation**. Microsoft Developer Network (MSDN). Retrieved from https://learn.microsoft.com/en-us/windows/win32/direct3d

2. **DirectX 9 SDK Documentation**. Microsoft Corporation. Available through DirectX SDK or MSDN documentation.

3. **3D Graphics Programming for Windows**. Programming Windows by Charles Petzold. Microsoft Press, 1998.

4. **Real-Time Rendering** by Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. A K Peters/CRC Press, 2008.

5. Online resources and discussions on Stack Overflow on common Direct3D initialization errors and 3D transformation troubleshooting.