Graphics Part II: Code Tutorial for Week 1

Dr P. Perakis

Preparation

The starting point for all of the exercises this week should be the example provided for you in `DirectX_Cube_Wireframe.zip`.

Introduction

The code in the example program currently just positions the cube at the origin and shows the front face of the cube. The world transformation matrix (`_worldTransformation` – defined in `DirectXApp.h`) is simply initialised to the Identity matrix and never changes.

For these exercises, you should modify the code so that it animates the cube. Most of your code changes will go in the Update method in `DirectXApp.cpp`, but you will almost certainly need to add one or more member variables to the class in `DirectXapp.h`.

To do these exercises, you will need to update the transformation matrix of the model `_worldTransformation` or multiply two or more matrices together and update the `_worldTransformation`. The transformation matrices can be generated by calls to the transformation creation static methods in the `SimpleMath Matrix` class.

Implementation

What you are suggested to do are:

Exercise 1a

Update the program so that the cube rotates one degree around the Z axis each frame.

Exercise 1b

Update the program so that the cube rotates one degree around the Y axis each frame.

Exercise 1c

Update the program so that the cube rotates one degree around the Y axis each frame and slowly moves towards and off the right-hand side of the window

Exercise 1d

Update the program so that the cube rotates one degree around the Z axis each frame but is always 2 units away from the origin (i.e. it does a wide circle around the Z axis).

If you have got this far, experiment with rotations around all axes and try the scaling transformation as well so that you are familiar with how they work. Try combing more than two matrices to see the impact.

1. Declaring the new variables

You were given a `DirectXApp` class as a starting point. You have to add the following variables for controlling the model's transformations:

```
int     _rotationAngle{ 0 };
float   _translationX{ 0.0f };
float   _scaleXYZ{ 1.0f };
```

2. Updating for the applied transformations

You were given as a starting point an empty `DirectXApp::Update()` function. You have to modify this function for transforming the model:

```
void DirectXApp::Update()
{
	Matrix RotationZ, RotationY, TranslationX, Scale;

	// This is where you would update world transformations

	//Exercise 1a
	RotationZ = Matrix::CreateRotationZ(_rotationAngle * XM_PI / 180.0f);
	_worldTransformation = RotationZ;

	//Exercise 1b
	RotationY = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
	_worldTransformation = RotationY;

	//Exercise 1c
	RotationY = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
	TranslationX = Matrix::CreateTranslation(_translationX,0,0);
	_worldTransformation = RotationY * TranslationX;

	//Exercise 1d
	RotationZ = Matrix::CreateRotationZ(_rotationAngle * XM_PI / 180.0f);
	TranslationX = Matrix::CreateTranslation(3.0f, 0, 0);
	_worldTransformation = TranslationX * RotationZ;

	//Exercise 1e
	RotationY = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
	RotationZ = Matrix::CreateRotationZ(_rotationAngle * XM_PI / 180.0f);
	TranslationX = Matrix::CreateTranslation(3.0f, 0, 0);
	_worldTransformation = RotationY * TranslationX * RotationZ ;

	//Exercise 1f
	RotationY = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
	RotationZ = Matrix::CreateRotationZ(_rotationAngle * XM_PI / 180.0f);
	TranslationX = Matrix::CreateTranslation(3.0f, 0, 0);
	Scale = Matrix::CreateScale(_scaleXYZ);
	//_worldTransformation = RotationY * TranslationX * RotationZ * Scale;
	_worldTransformation = Scale * RotationY * TranslationX * RotationZ;


	_rotationAngle = (_rotationAngle + 1) % 360;
	_translationX = _translationX + 0.01f;
	_scaleXYZ = _scaleXYZ - 0.001f;
	if (_scaleXYZ < 0.001) _scaleXYZ = 1.0f;
}
```

Note that you have to keep in the code only the statements for the specific Exercise you want to run.

Exercise 2

For this exercise, modify the program so that it draws an additional cube above the first one. The top cube show rotate around the Y axis in one direction and the bottom cube should rotate around the Y axis in the opposite direction.

3. Declaring the new variables

You were given a `DirectXApp` class as a starting point. You have to declare the following variables for keeping the two models' transformations:

```
Matrix          _worldTransformation1;
Matrix          _worldTransformation2;
```

4. Updating for the applied transformations

You were given as a starting point an empty `DirectXApp::Update()` function. You have to modify this function for transforming the two models.

```
void DirectXApp::Update()
{
	Matrix RotationZ, RotationY, TranslationY, Scale;

	// This is where you would update world transformations

	//Exercise 2

	//Cube 1
	RotationY = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
	_worldTransformation1 = RotationY;

	//Cube 2
	RotationY = Matrix::CreateRotationY(-0.5f * _rotationAngle * XM_PI / 180.0f);
	TranslationY = Matrix::CreateTranslation(0,-2.0f,0);
	_worldTransformation2 = RotationY * TranslationY;


	_rotationAngle = (_rotationAngle + 1) % 720;

}
```

5. Modifications for rendering the two models

You were given as a starting point the `DirectXApp::Render()` function. You have to modify this function accordingly, for rendering the two models separately:

```
void DirectXApp::Render()
{
	const float clearColour[] = { 0.0f, 0.0f, 0.0f, 1.0f };
	_deviceContext->ClearRenderTargetView(_renderTargetView.Get(), clearColour);
	_deviceContext->ClearDepthStencilView(_depthStencilView.Get(),
D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

	_viewTransformation = XMMatrixLookAtLH(_eyePosition, _focalPointPosition,
_upVector);
	_projectionTransformation = XMMatrixPerspectiveFovLH(XM_PIDIV4,
static_cast<float>(GetWindowWidth()) / GetWindowHeight(), 1.0f, 100.0f);


	Matrix completeTransformation;
	CBuffer constantBuffer;
	UINT stride;
	UINT offset;

	//Cube 1
	// Calculate the world x view x projection transformation
	completeTransformation = _worldTransformation1 * _viewTransformation *
_projectionTransformation;

	constantBuffer.WorldViewProjection = completeTransformation;
```

```
	// Update the constant buffer. Note the layout of the constant buffer must 
match that in the shader
	_deviceContext->VSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());
	_deviceContext->UpdateSubresource(_constantBuffer.Get(), 0, 0, &constantBuffer, 
0, 0);

	// Now render the cube
	// Specify the distance between vertices and the starting point in the vertex 
buffer
	stride = sizeof(Vertex);
	offset = 0;
	// Set the vertex buffer and index buffer we are going to use
	_deviceContext->IASetVertexBuffers(0, 1, _vertexBuffer.GetAddressOf(), &stride, 
&offset);
	_deviceContext->IASetIndexBuffer(_indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);

	// Specify the layout of the polygons (it will rarely be different to this)
	_deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

	// Specify the layout of the input vertices.  This must match the layout of the 
input vertices in the shader
	_deviceContext->IASetInputLayout(_layout.Get());

	// Specify the vertex and pixel shaders we are going to use
	_deviceContext->VSSetShader(_vertexShader.Get(), 0, 0);
	_deviceContext->PSSetShader(_pixelShader.Get(), 0, 0);

	// Specify details about how the object is to be drawn
	_deviceContext->RSSetState(_rasteriserState.Get());

	// Now draw the object
	_deviceContext->DrawIndexed(ARRAYSIZE(indices), 0, 0);


	//Cube 2
	// Calculate the world x view x projection transformation
	completeTransformation = _worldTransformation2 * _viewTransformation * 
_projectionTransformation;

	constantBuffer.WorldViewProjection = completeTransformation;

	// Update the constant buffer. Note the layout of the constant buffer must 
match that in the shader
	_deviceContext->VSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());
	_deviceContext->UpdateSubresource(_constantBuffer.Get(), 0, 0, &constantBuffer, 
0, 0);

	// Now render the cube
	// Specify the distance between vertices and the starting point in the vertex 
buffer
	stride = sizeof(Vertex);
	offset = 0;
	// Set the vertex buffer and index buffer we are going to use
	_deviceContext->IASetVertexBuffers(0, 1, _vertexBuffer.GetAddressOf(), &stride, 
&offset);
	_deviceContext->IASetIndexBuffer(_indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);

	// Specify the layout of the polygons (it will rarely be different to this)
	_deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

	// Specify the layout of the input vertices.  This must match the layout of the 
input vertices in the shader
```

```
        _deviceContext->IASetInputLayout(_layout.Get());

        // Specify the vertex and pixel shaders we are going to use
        _deviceContext->VSSetShader(_vertexShader.Get(), 0, 0);
        _deviceContext->PSSetShader(_pixelShader.Get(), 0, 0);

        // Specify details about how the object is to be drawn
        _deviceContext->RSSetState(_rasteriserState.Get());

        // Now draw the object
        _deviceContext->DrawIndexed(ARRAYSIZE(indices), 0, 0);


        // Update the window
        ThrowIfFailed(_swapChain->Present(0, 0));
}
```

Note that you keep the same indices structure when calling the rendering function `_deviceContext->DrawIndexed()`. *You are just applying two different model transformations using the* `completeTransformation` *parameter in the constant buffer.*

Exercise 3

Experiment with changing the eye position (`_eyePosition`) and look at position (`_focalPointPosition`) vectors to change the camera position. For example, can you change the camera so that it:

a) Looks diagonally down at the cubes from the front, but at a higher position
b) Looks vertically down on to the two rotating cubes.

6. Declaring the new variables

You were given a `DirectXApp` class as a starting point. You have to declare the following variables for keeping the camera's transformations:

```
int             _cameraAngleX{ 0 };
float           _cameraDistance{ 10.0f };
```

7. Updating for the applied transformations

You were given as a starting point an empty `DirectXApp::Update()` function. You have to modify this function for transforming the camera:

```
void DirectXApp::Update()
{

        //Exercise 3a
        _eyePosition.y = _eyePosition.y + 0.01f;
        _focalPointPosition.y = _focalPointPosition.y + 0.01f;
        if (_eyePosition.y > 10.0f)
        {
                _eyePosition.y = -10.0f;
                _focalPointPosition.y = -10.0f;
        }

        //Exercise 3b
        _cameraAngleX = (_cameraAngleX + 1) % 360;
        if (_cameraAngleX > 90) _cameraAngleX = -90;
```

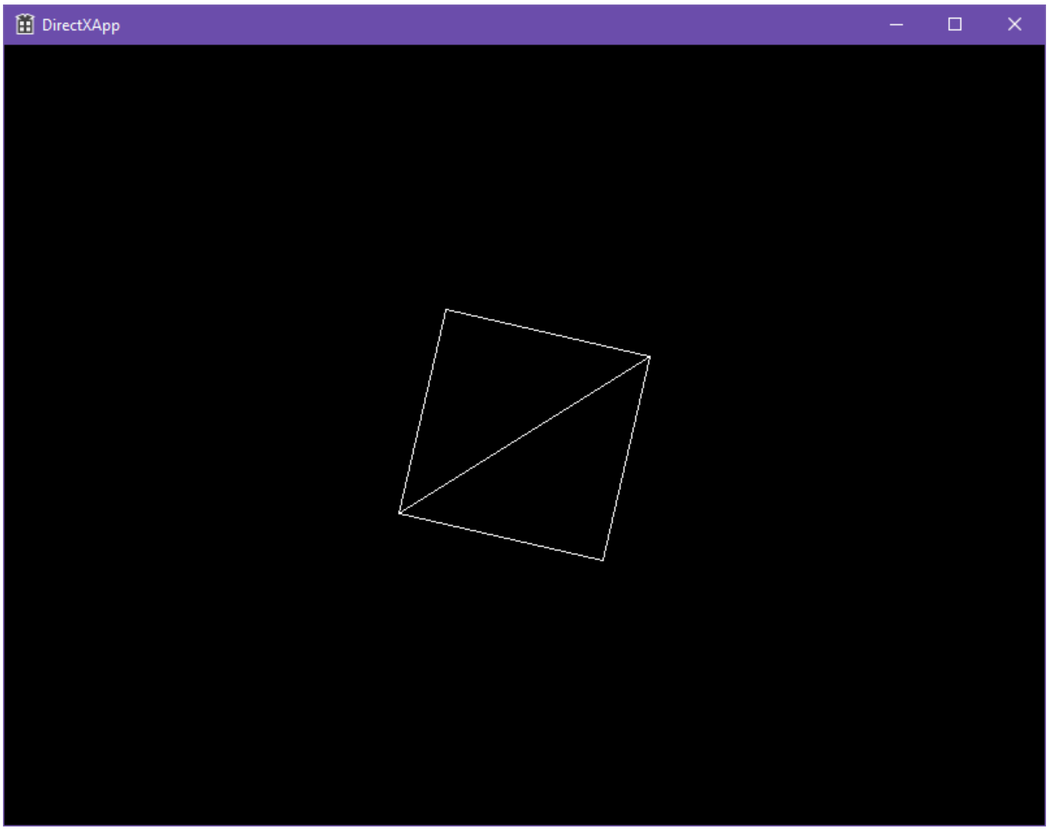
```
        //_cameraAngleX = 60;
        _cameraDistance = 15.0f;
        _eyePosition.y = _cameraDistance * sin(_cameraAngleX * XM_PI / 180.0f);
        _eyePosition.z = -_cameraDistance * cos(_cameraAngleX * XM_PI / 180.0f);

}
```

Note that you have to keep in the code only the statements for the specific Exercise you want to run. For Exercise 3a the camera moves from y = -10 up to y = +10, having a horizontal direction. For Exercise 3b the camera keeps looking at the origin and moves from an angle of -90 degrees, showing the bottom of the model, up to +90 degrees, showing the top of the model.

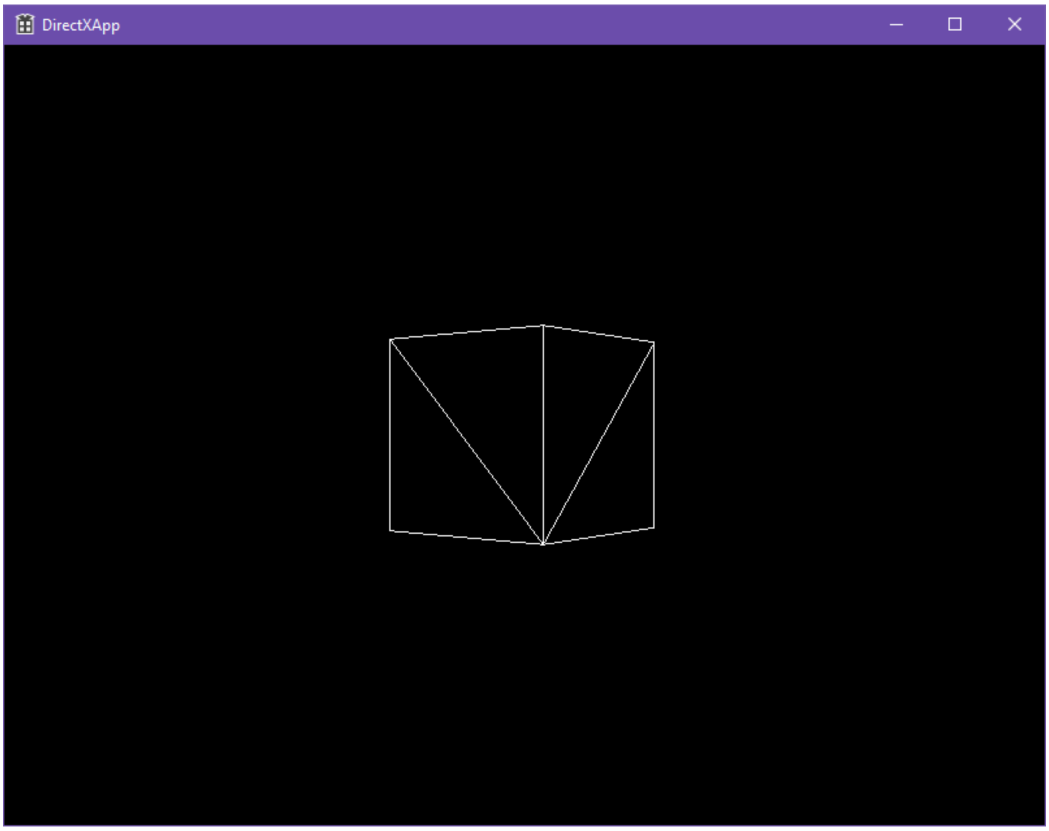
8. The output of the executable

After compiling and running the executable, the application window should look like this:

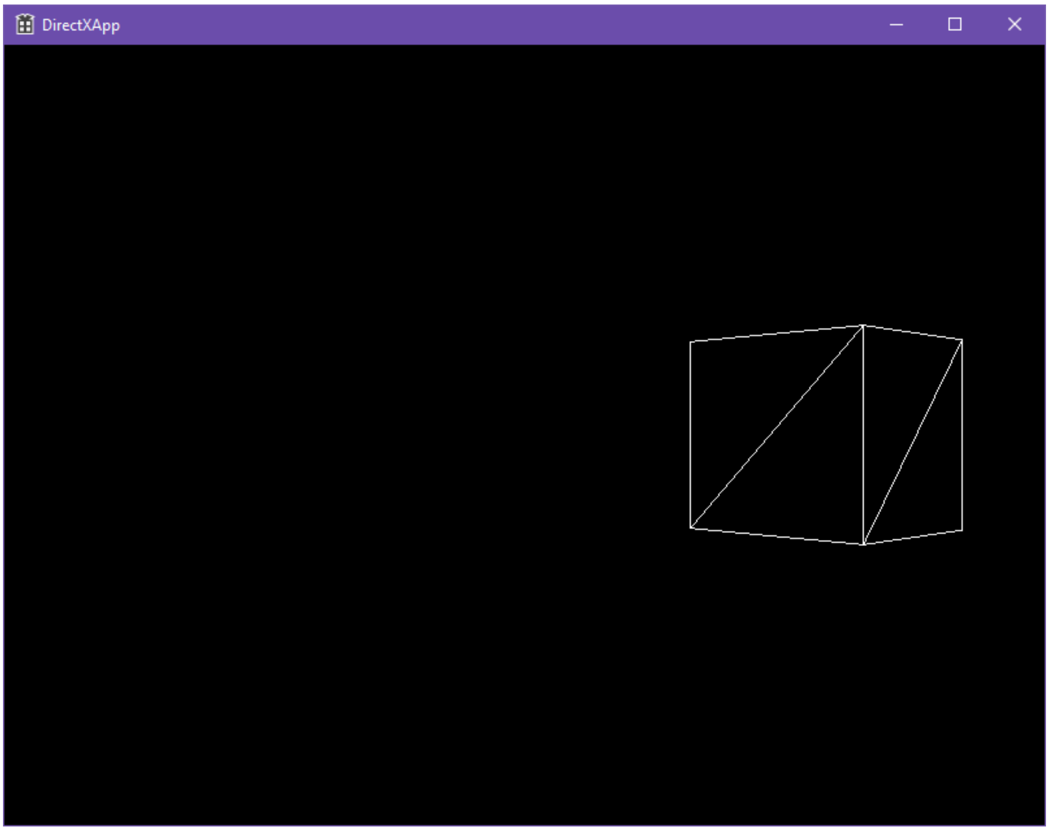
Exercise 1a



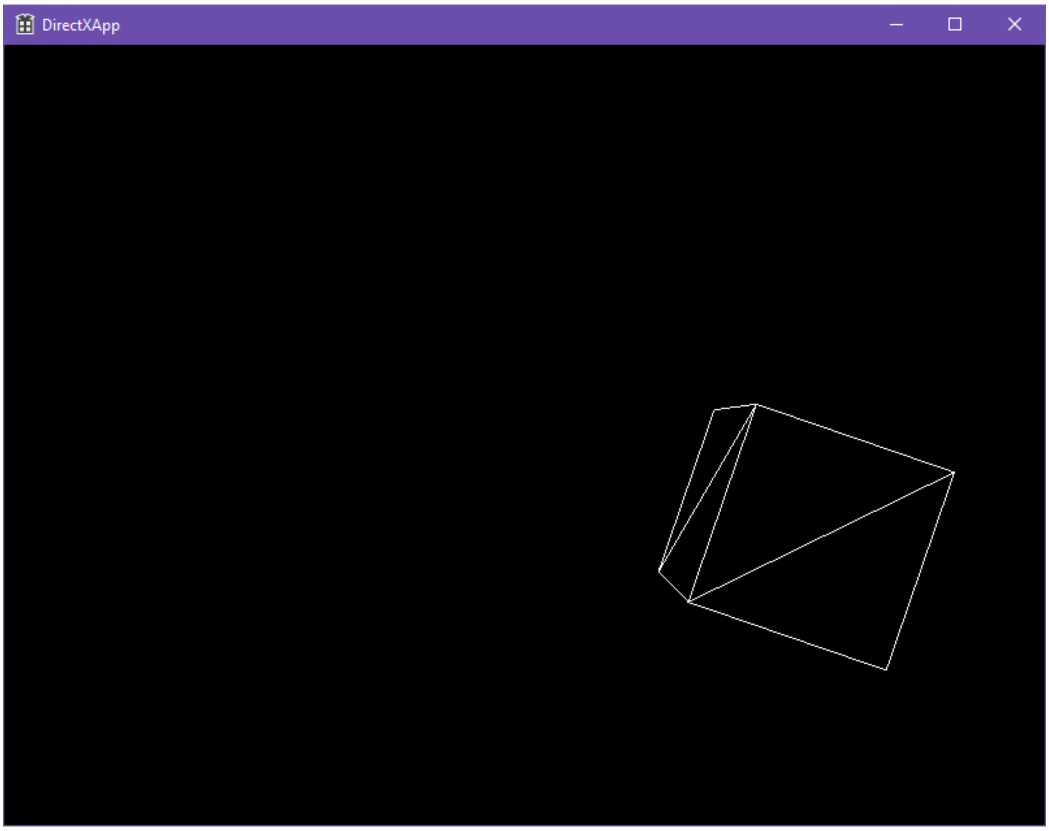
Exercise 1b



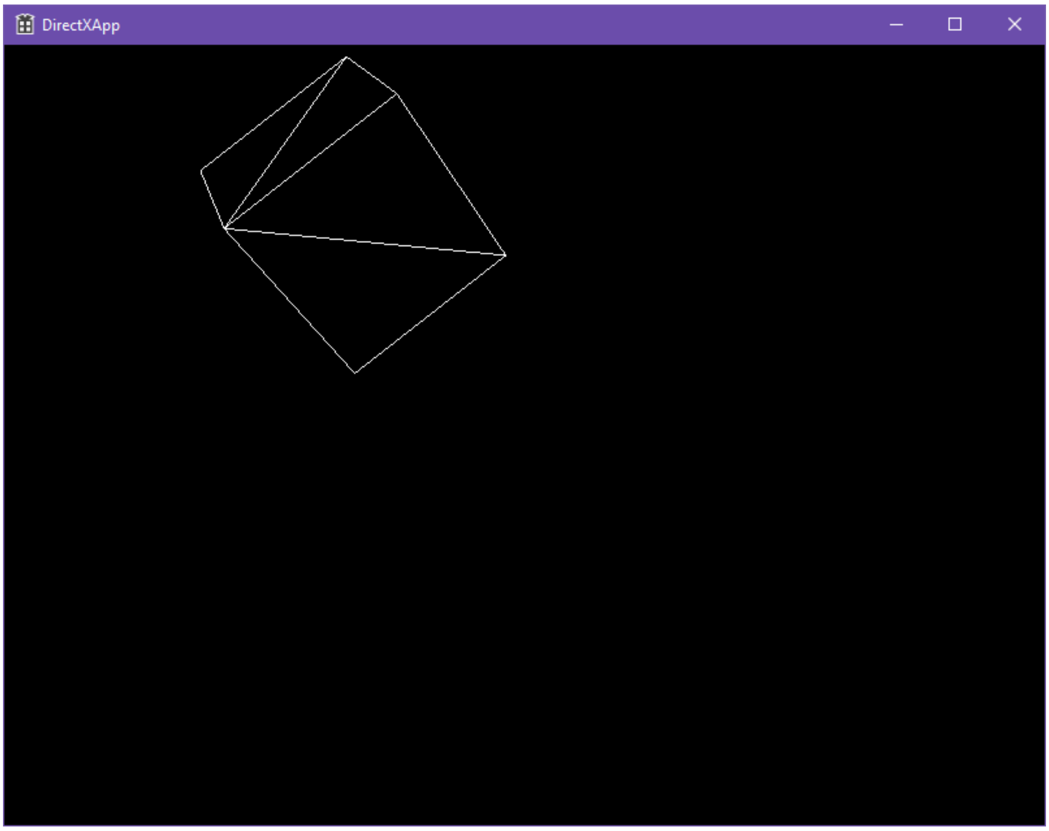
Exercise 1c



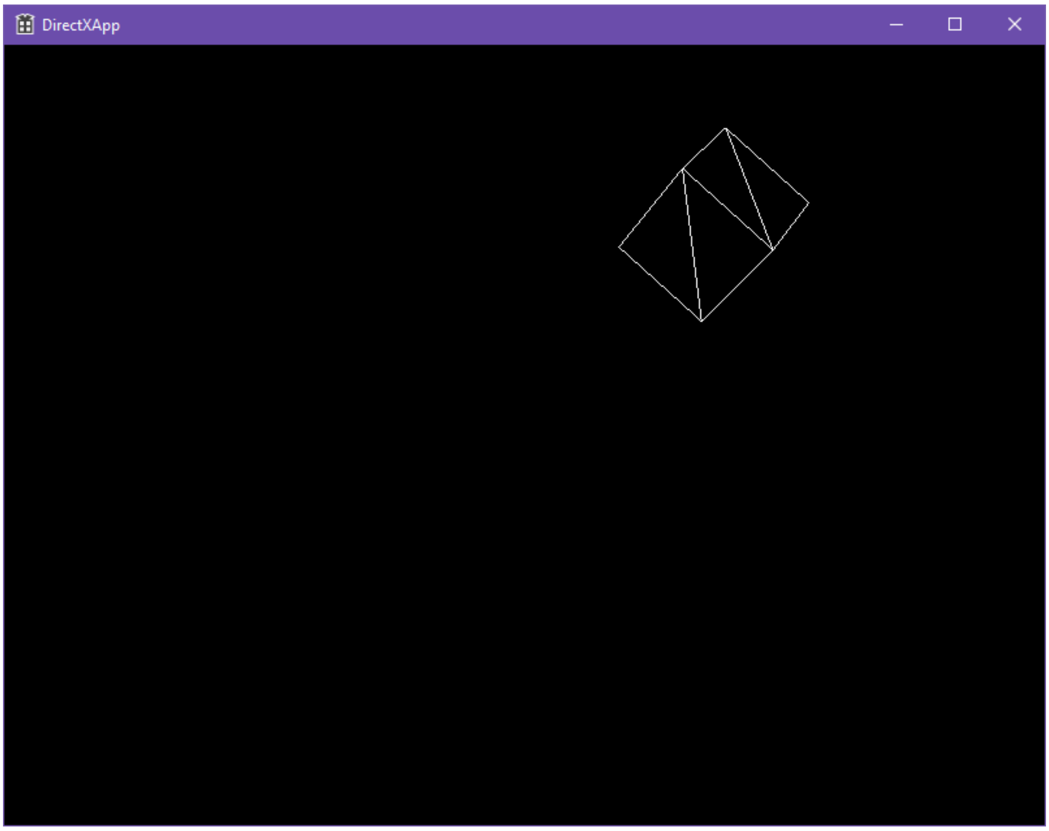
Exercise 1d



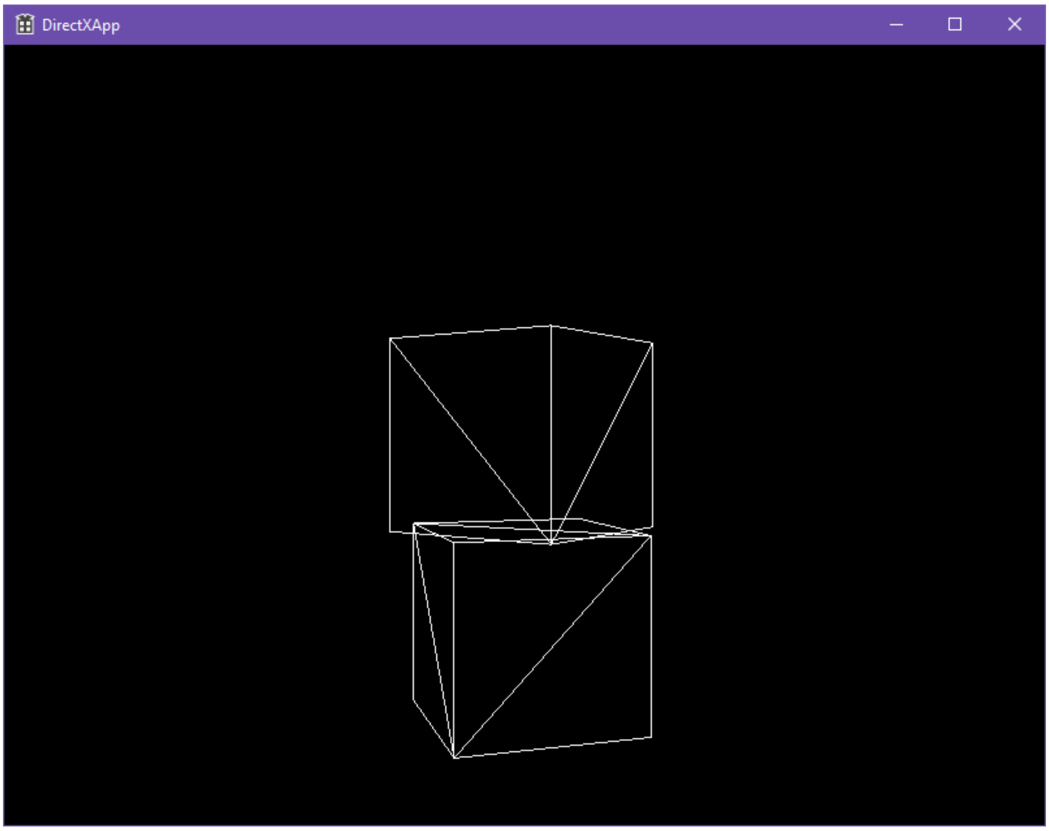
Exercise 1e



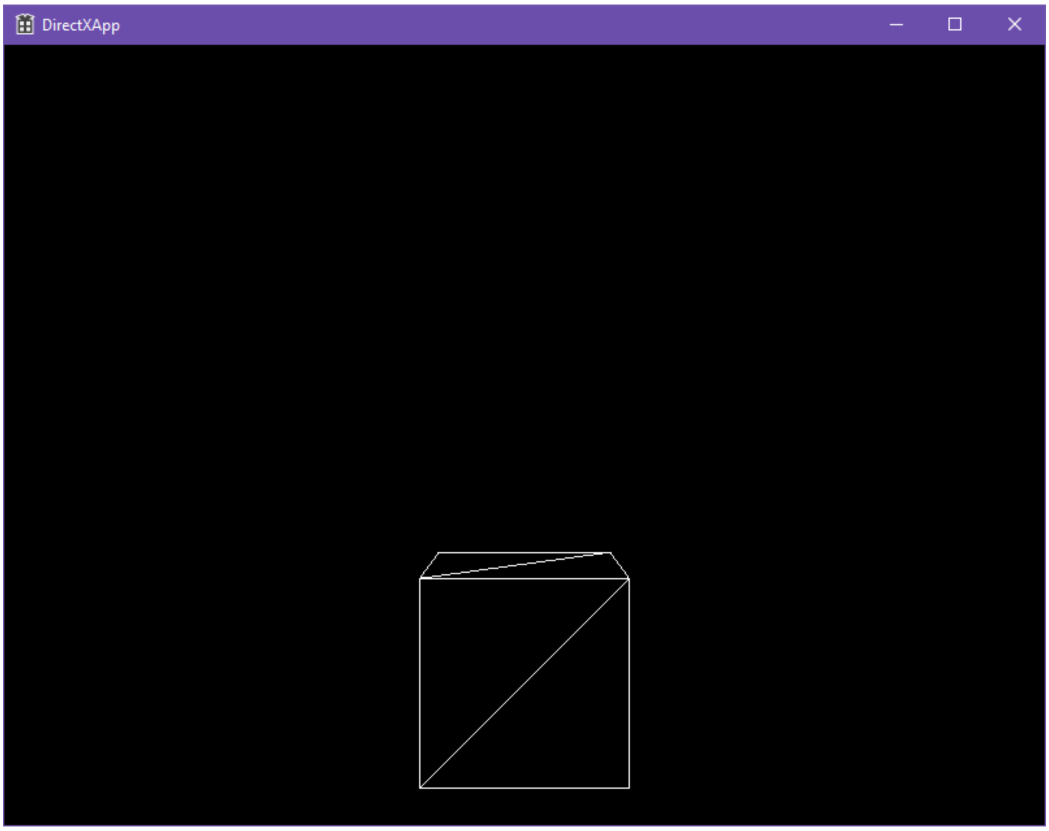
Exercise 1f



Exercise 2



Exercise 3a



Exercise 3b