Realtime Graphics Lab Book

# Week 1 – Lab B

Date: 13rd Oct 2023

## Exercise 1

### Question: Modify the vertex list of the cube to draw a hexagonal cylinder.

### Solution:

const int divisionCount = 6;

const int vertexCount = divisionCount \* 2 + 2; // Number of vertices plus two centers

const int indexCount = (divisionCount \* 6) + (divisionCount \* 2 \* 3);

SimpleVertex vertices[divisionCount \* 2 + 2]; // Number of vertices plus two centers

XMFLOAT4 colors[] = {

XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f), // Blue

XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f), // Green

XMFLOAT4(0.0f, 1.0f, 1.0f, 1.0f), // Cyan

XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f), // Red

XMFLOAT4(1.0f, 0.0f, 1.0f, 1.0f), // Magenta

XMFLOAT4(1.0f, 1.0f, 0.0f, 1.0f), // Yellow

};

vertices[0] = { XMFLOAT3(0.0f, 1.0f, 0.0f), XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f) };

vertices[divisionCount + 1] = { XMFLOAT3(0.0f, -1.0f, 0.0f), XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f) };

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

float angle = XM\_2PI \* i / static\_cast<float>(divisionCount);

float x = cos(angle);

float z = sin(angle);

vertices[i + 1] = { XMFLOAT3(x, 1.0f, z), colors[i % 6] };

vertices[divisionCount + i + 2] = { XMFLOAT3(x, -1.0f, z), colors[i % 6] };

}

int index = 0;

// Top face indices

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

indices[index++] = 0;

indices[index++] = (i + 1) % divisionCount + 1;

indices[index++] = i + 1;

}

// Bottom face indices

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

indices[index++] = divisionCount + 1;

indices[index++] = divisionCount + 2 + i;

indices[index++] = divisionCount + 2 + (i + 1) % divisionCount;

}

// Side face indices

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

int top1 = i + 1;

int top2 = (i + 1) % divisionCount + 1;

int bottom1 = divisionCount + 2 + i;

int bottom2 = divisionCount + 2 + (i + 1) % divisionCount;

// First triangle

indices[index++] = top1;

indices[index++] = bottom2;

indices[index++] = bottom1;

// Second triangle

indices[index++] = top1;

indices[index++] = top2;

indices[index++] = bottom2;

}

### Sample output:

1. A colorful prism on a blue background

   Description automatically generated

2. A colorful striped object on a blue background

Description automatically generated

### Reflection:

*To create this cylinder we can simply create the vertices list in runtime and add to it based on some algorithm, so the main factor here is divisionCount, and then we can draw the cylinder with correct front facing faces.*

## Exercise 2

Modify the vertex list in indices[] or modify the parameters in the DrawIndexed( ) to draw: (1) two  
triangles; (2) one face of the cube; (3) the four walls of the cube  
You may notice that only one face of the triangle and square face being drawn. This is because by  
default, the back face of the cube is culled out. You can specify the cull mode to be none by filling.

### Solution:

for (int i = 0; i <= rows; ++i)

{

for (int j = 0; j <= columns; ++j)

{

float x = static\_cast<float>(j) / columns \* 2;

float y = 0.0f; // Flat grid on the XZ plane

float z = static\_cast<float>(i) / rows \* 2;

XMFLOAT4 color = XMFLOAT4(x, y, z, 1.0f); // Color based on position

vertices.push\_back({ XMFLOAT3(x, y, z), color});

}

}

// Create index buffer

std::vector<WORD> indices;

// Generate indices for triangle strips

for (int i = 0; i <= rows; ++i)

{

for (int j = 0; j <= columns; ++j)

{

indices.push\_back((i + 1) \* (columns)+j);

indices.push\_back(i \* (columns) + j);

}

if(rows - 1 > i)

indices.push\_back(-1);

}

### Sample output:

A blue and pink grid

Description automatically generated

### Reflection:

*We can define two variables, rows and columns, which represent the number of strips in the grid.*

## Exercise 3

### Specify different heights at different grid points for the 3D grid you created in Exercise 6 to create a terrain triangle mesh.

### Solution:

int count = 0;

for (int i = 0; i <= rows; ++i)

{

for (int j = 0; j <= columns; ++j)

{

float x = static\_cast<float>(j) / columns \* 5;

rng\_type::result\_type const seedval = count++; // get this from somewhere

rng.seed(seedval);

rng\_type::result\_type random\_number = udist(rng) % 200;

// Normalize random\_number between 0 and 1

double normalizedNumber = random\_number / 200.0 \* 5;

float y = 0.0f; // Flat grid on the XZ plane

float z = static\_cast<float>(i) / rows \* 5;

XMFLOAT4 color = XMFLOAT4(x, y, z, 1.0f); // Color based on position

vertices.push\_back({ XMFLOAT3(x, normalizedNumber, z), color});

}

}

### Sample output:

A grid of lines on a blue background

Description automatically generated

### Reflection:

*To create different heights, we can randomize the Y values with a maximum height of 3 for example.*

## Exercise 4

### Modify the parameter in IASetPrimitiveTopology( ) and indices[] to draw: 1. A list of points corresponding to the cube’s eight vertices. 2. The 12 edges of the cube (not as a wireframe triangle mesh).

### Solution:

RenderItem it;

auto box = gen.CreateSphere(1, 5, 5);

it.Geo = &(box);

it.IndexCount = it.Geo->GetIndices16().size();

g\_RenderItems.push\_back(it);

RenderItem it2;

auto spher = gen.CreateGrid(8, 8, 10, 10);

it2.Geo = &(spher);

it2.World = XMMatrixIdentity();

it2.World \*= XMMatrixTranslation(-0.0f, -1.0f, 0.0f);

it2.IndexCount = it2.Geo->GetIndices16().size();

g\_RenderItems.push\_back(it2);

RenderItem it3;

auto cyl = gen.CreateCylinder(1,0.3f,3, 8,8);

it3.Geo = &(cyl);

it3.World = XMMatrixIdentity();

it3.World \*= XMMatrixTranslation(-3.0f, -0.0f, 0.0f);

it3.IndexCount = it3.Geo->GetIndices16().size();

g\_RenderItems.push\_back(it3);

for (auto& tt : g\_RenderItems) {

D3D11\_BUFFER\_DESC bd = {};

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof(GeometryGenerator::Vertex) \* tt.Geo->Vertices.size();

bd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

bd.CPUAccessFlags = 0;

D3D11\_SUBRESOURCE\_DATA InitData = {};

InitData.pSysMem = tt.Geo->Vertices.data();

hr = g\_pd3dDevice->CreateBuffer(&bd, &InitData, &tt.verticesBuffer);

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof(GeometryGenerator::uint16) \* tt.Geo->GetIndices16().size();

bd.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

bd.CPUAccessFlags = 0;

InitData.pSysMem = tt.Geo->GetIndices16().data();

hr = g\_pd3dDevice->CreateBuffer(&bd, &InitData, &tt.indicesBuffer);

}

### Sample Output:

A wireframe of a building

Description automatically generated with medium confidence

### Reflection:

*We can create the render item struct that would hold the vertices and indices buffer along with other necessary information, then we can iterate over every object to render it and set the correct corresponding buffers.*

## Exercise 5

### Draw two wireframe cubes. There are different ways of achieving this. One simple way is to pass a different world matrix to the vertex shader.

### Solution:

g\_World \*= XMMatrixTranslation(-2, 2, -1);

g\_World \*= XMMatrixScaling(0.5, 0.5, 0.5);

cb.mWorld = XMMatrixTranspose(g\_World);

g\_pImmediateContext->UpdateSubresource(g\_pConstantBuffer, 0, nullptr, &cb, 0, 0);

g\_pImmediateContext->DrawIndexed(36, 0, 0); // 36 vertices needed for 12 triangles in a triangle list

### Sample output:

A colorful cube on a blue background

Description automatically generated

### Reflection:

*We can simply use the same vertices and indices because the cube model didn’t change, so we change the world matrix to translate the cube again after drawing it the first time and scale it down a little bit. And then we draw it again so it looks like we have two cubes now on the screen by recycling the same vertices and indices.*

## Exercise 5

### Draw the cube as triangle strips by setting primitive topology as D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLESTRIP.

### Solution:

WORD wallIndices[] =

{

// Front face

0, 1, 4, 5,

// Right face

1, 2, 5, 6,

// Back face

2, 3, 6, 7,

// Left face

3, 0, 7, 4

};

WORD topIndices[] =

{

3, 2, 0, 1

};

WORD bottomIndices[] =

{

4, 5, 7, 6

};

### Sample output:

A colorful cube on a blue background

Description automatically generated

### Reflection:

*We cannot render the whole cube using triangle strip in one go, we can separate the cube into three triangle strips instead, the walls of the cube are a triangle strip, the bottom and the top faces are two individual triangle strips.  
So then we must separate our indices into 3 different arrays, and then we will draw each individual array as a triangle strip*