AGP Report
-Luis Cebrián Chuliá-

Std. num 500747256

1-Primitives:

Category: Advanced - Renders dynamic primitive topology on the GPU

In this assignment we draw a lot of things because we thought that we needed to complete all the categories to score a high grade.

First we created two pyramids manually inserting the coordinates of the vertices and indices.

```
Vertex verticesArray[] =
{
    //First pyramid
    { XMFLOAT3(-1.0f, -1.0f, +1.0f), (const float*)&Colors::White   },
    { XMFLOAT3(+1.0f, -1.0f, +1.0f), (const float*)&Colors::Black   },
    { XMFLOAT3(+1.0f, -1.0f, -1.0f), (const float*)&Colors::Red     },
    { XMFLOAT3(-1.0f, -1.0f, -1.0f), (const float*)&Colors::Green   },
    { XMFLOAT3(0.0f, +1.0f, 0.0f), (const float*)&Colors::Blue    },
    //Second pyramid
    { XMFLOAT3(-1.0f, +3.0f, +1.0f), (const float*)&Colors::White   },
    { XMFLOAT3(+1.0f, +3.0f, +1.0f), (const float*)&Colors::Black },
    { XMFLOAT3(+1.0f, +3.0f, -1.0f), (const float*)&Colors::Red },
    { XMFLOAT3(-1.0f, +3.0f, -1.0f), (const float*)&Colors::Green}

};
```

Then we create a sphere using the geometry generator

```
GeometryGenerator::MeshData sphere;
GeometryGenerator::MeshData octagonalPrism;

GeometryGenerator geoGen;
geoGen.CreateSphere(2.0f, 20, 20, sphere);
createOctagonalPrism(1.0f, 2.0f, octagonalPrism);
```

To create the octagonal prism we defined our own function that deals with it. This function could create a N-prism by changing the slice variables, but we set this variable to 8 to always be a octagonal prism. In order to create it we first define the top and bottom vertices of the prism.

```
UINT sliceCount = 8;

float dTheta = 2.0f*XM_PI / sliceCount;
//Build the vertex buffer
//Bottom
float y = -0.5f*height;
for (UINT i = 0; i <= sliceCount; ++i)
{
    GeometryGenerator::Vertex vertex;

    float c = cosf(i*dTheta);
    float s = sinf(i*dTheta);

    vertex.Position = XMFLOAT3(radius*c, y, radius*s);
    meshData.Vertices.push_back(vertex);
}

//Top
y = 0.5f*height;
for (UINT i = 0; i <= sliceCount; ++i)
{
    GeometryGenerator::Vertex vertex;

    float c = cosf(i*dTheta);
    float s = sinf(i*dTheta);

    vertex.Position = XMFLOAT3(radius*c, y, radius*s);
    meshData.Vertices.push_back(vertex);
}
```

Then we build the indices:

```
//Build the indices
for (UINT i = 0; i < 1; ++i)
{
    for (UINT j = 0; j <= sliceCount; ++j)
    {
        meshData.Indices.push_back(i*sliceCount + j);
        meshData.Indices.push_back((i + 1)*sliceCount + j);
        meshData.Indices.push_back((i + 1)*sliceCount + j + 1);

        meshData.Indices.push_back(i*sliceCount + j);
        meshData.Indices.push_back((i + 1)*sliceCount+ j + 1);
        meshData.Indices.push_back(i*sliceCount + j + 1);
    }
}
```

And then, we just need to add the center vertex of the top and bottom of the prism and the indices to correctly render the prism.

Once we have all the indices and vertices of all the objects we just need to define the index and vertex buffers.To do so we need to reserve enough memory to all the vertices and indices.

That’s why we retrieve the size of the vertex and index and multiply them but the number of indices and vertices.

```
D3D11_BUFFER_DESC vbd;
vbd.Usage = D3D11_USAGE_IMMUTABLE;
vbd.ByteWidth = sizeof(Vertex) * totalVertexCount;
vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags = 0;
vbd.MiscFlags = 0;
vbd.StructureByteStride = 0;
D3D11_SUBRESOURCE_DATA vinitData;
vinitData.pSysMem = &vertices[0];
HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mBoxVB));
```

```
D3D11_BUFFER_DESC ibd;
ibd.Usage = D3D11_USAGE_IMMUTABLE;
ibd.ByteWidth = sizeof(UINT) * indices.size();
ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
ibd.CPUAccessFlags = 0;
ibd.MiscFlags = 0;
ibd.StructureByteStride = 0;
D3D11_SUBRESOURCE_DATA iinitData;
iinitData.pSysMem = &indices[0];
HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mBoxIB));
```

We render multiple spheres but we only created one in the geometry buffer. We do so using instancing. We define a matrix containing the rotation, scale and displacement of each sphere we want to render.

```
XMMATRIX centerSphereScale = XMMatrixScaling(2.0f, 2.0f, 1.0f);
XMMATRIX centerSphereOffset = XMMatrixTranslation(0.0f, 1.0f, 0.0f);
XMStoreFloat4x4(&mCenterSphere, XMMatrixMultiply(centerSphereScale, centerSphereOffset));

XMMATRIX leftSphereScale = XMMatrixScaling(0.5f, 0.5f, 0.5f);
XMMATRIX leftSphereOffset = XMMatrixTranslation(-3.0f, 4.0f, 0.0f);
XMStoreFloat4x4(&mLeftSphere, XMMatrixMultiply(leftSphereScale, leftSphereOffset));

XMMATRIX rightSphereScale = XMMatrixScaling(0.5f, 0.5f, 0.5f);
XMMATRIX rightSphereOffset = XMMatrixTranslation(3.0f, 4.0f, 0.0f);
XMStoreFloat4x4(&mRightSphere, XMMatrixMultiply(rightSphereScale, rightSphereOffset));

XMMATRIX prismOffset = XMMatrixTranslation(0.0f, -4.0f, 0.0f);
XMStoreFloat4x4(&mOctagonalPrism, prismOffset);
```

And then use this matrix before rendering in our drawScene method.

```
world = XMLoadFloat4x4(&mLeftSphere);
mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&(world*view*proj)));
mTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(mSphereIndexCount, 36, 9);

world = XMLoadFloat4x4(&mRightSphere);
mfxWorldViewProj->SetMatrix(reinterpret_cast<float*>(&(world*view*proj)));
mTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(mSphereIndexCount, 36, 9);
```

Note that we draw the spheres using the same information in the DrawIndexed(...) call.

We also change between render states in order to display different effects. We change between the default state (solid with face culling) and the wire frame state. To implement the wireframe state we need to fill a descriptor:

```
D3D11_RASTERIZER_DESC wireframeDesc;
ZeroMemory(&wireframeDesc, sizeof(D3D11_RASTERIZER_DESC));
wireframeDesc.FillMode = D3D11_FILL_WIREFRAME;
wireframeDesc.CullMode = D3D11_CULL_NONE;
wireframeDesc.FrontCounterClockwise = false;
wireframeDesc.DepthClipEnable = true;
```

and set the render state before drawing what we want in wire frame mode:

```
md3dImmediateContext->RSSetState(mWireframeRS);
```

In order to use tessellation. We need to change first the primitive topology in the drawScen() method to:

```
md3dImmediateContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST);
```

Because we now draw patches rather than triangles, we need to specify the hull shader, which will generate new vertices given a tesselation factor. We do so by implementing in the shader:

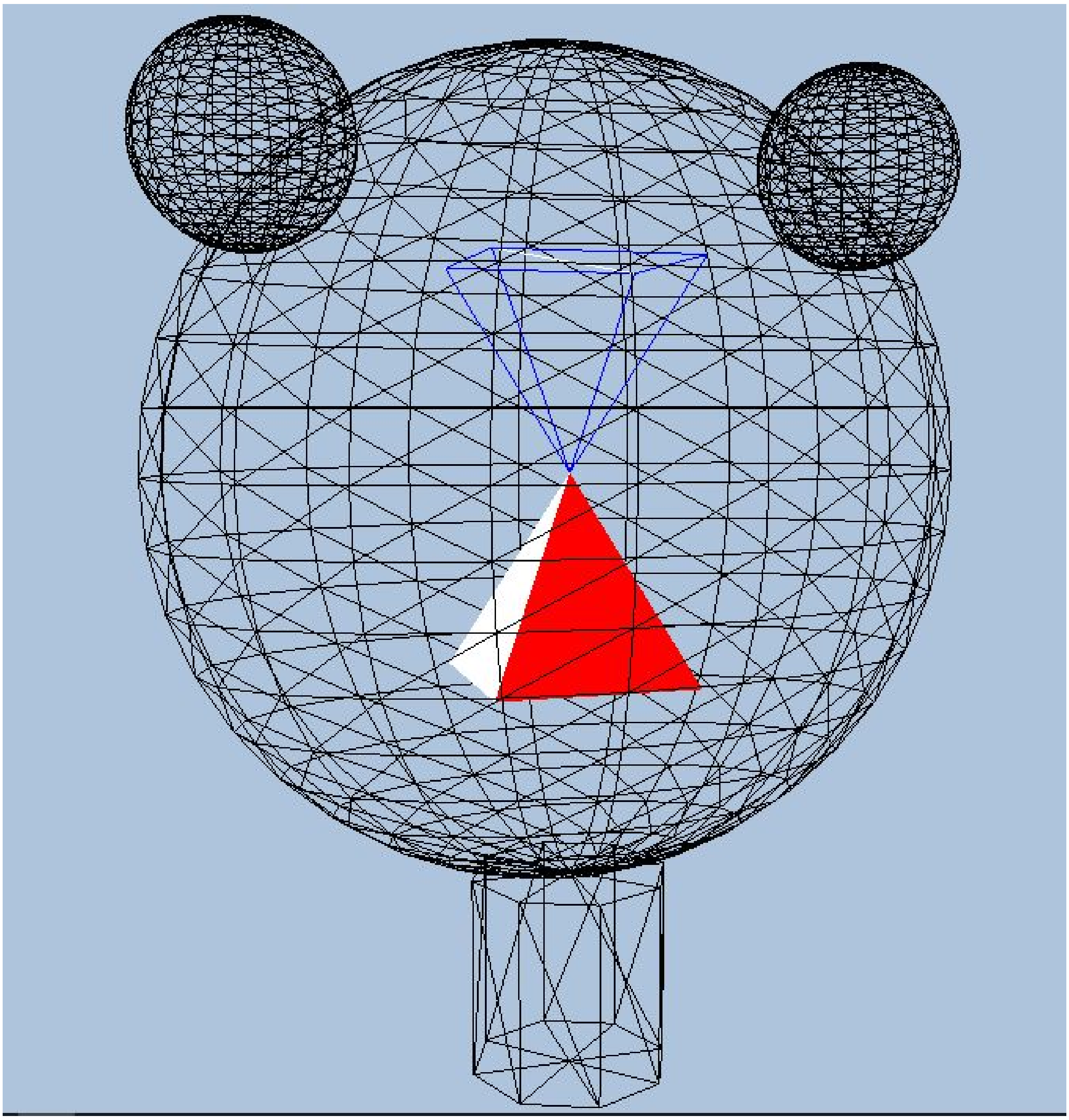
```
PatchTess ConstantHS(InputPatch<VertexOut, 3> patch, uint patchID : SV_PrimitiveID)
{
    PatchTess pt;

    pt.EdgeTess[0] = tessFactor;
    pt.EdgeTess[1] = tessFactor;
    pt.EdgeTess[2] = tessFactor;

    pt.InsideTess = 1;

    return pt;
}

struct HullOut
{
    float3 PosL : POSITION;
    float4 Color : COLOR;
};

[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("ConstantHS")]
[maxtessfactor(64.0f)]
HullOut HS(InputPatch<VertexOut, 3> p,
    uint i : SV_OutputControlPointID,
    uint patchId : SV_PrimitiveID)
{
    HullOut hout;

    hout.PosL = p[i].PosL;
    hout.Color = p[i].Color;

    return hout;
}

```

The parameter tessFactos is a variable we manually can set by pressing 1 2 and 3 to change the number of new vertices generated within the shader. To do so we needed to define a new variable and link it to the application.(This is explained in more detail in the shading assignment)

Because we enabled tessellation we also need to create the domain shader which will deal with the vertices generated by the tessellator

```
struct DomainOut
{
    float4 PosH : SV_POSITION;
    float4 Color: COLOR;
};

[domain("tri")]
DomainOut DS(PatchTess patchTess,
    float3 abc: SV_DomainLocation,
    const OutputPatch<HullOut, 3> tri)
{
    DomainOut dout;

    float3 p = abc.x * tri[0].PosL + abc.y * tri[1].PosL + abc.z*tri[2].PosL;
    dout.PosH = mul(float4(p, 1.0f), gWorldViewProj);
    dout.Color = tri[0].Color;
    return dout;

}
```

The result of all explained above is:



2-Models:

Category: Advanced - Load model data in an industry standard format using open asset library.

**Note: The geometry can be loaded from every other format the same way we did, but because we tried to load a specific model with its textures, the way we loaded the textures will be different for every other format.*

This code is a modification of the code posted
http://www.gamedev.net/topic/652669-direct3d-11-load-3d-models-assimp/
and
http://www.gamedev.net/topic/653428-directx-11-file-importing/
So it will be very similar to mine.

We implemented a loader for the blender format.* We created our own class to do it, so we will go directly to the relevant code.

```
void BlenderModel::LoadModel(const std::string & filename,TextureMgr* mTexMgr)
{
    Assimp::Importer imp;

    const aiScene* pScene = imp.ReadFile(filename,
        aiProcess_CalcTangentSpace |
        aiProcess_Triangulate |
        aiProcess_GenSmoothNormals |
        aiProcess_SplitLargeMeshes |
        aiProcess_ConvertToLeftHanded |
        aiProcess_SortByPType |
        aiProcess_PreTransformVertices);

    if (pScene == NULL)
        printf(imp.GetErrorString());

    std::vector<Vertex::Basic32> vertices;
    std::vector<USHORT> indices;
    std::vector<MeshGeometry::Subset> subsets;
```

The first thing we need to do in our LoadModel(...) function is to load the model using assimp and define a vector for the vertices, indices and subsets.

Then, we will have to iterate over the meshes of the model and extract its vertices, indices, subsets, materials and textures. We do it this way:

```
    for (UINT i = 0; i < pScene->mNumMeshes; i++)
    {
        aiMesh* mesh = pScene->mMeshes[i];
        aiMaterial* material = pScene->mMaterials[mesh->mMaterialIndex];
        MeshGeometry::Subset subset;

        subset.VertexCount = mesh->mNumVertices;
        subset.VertexStart = vertices.size();
        subset.FaceStart = indices.size() / 3;
        subset.FaceCount = mesh->mNumFaces;
        subset.Id = mesh->mMaterialIndex;
        mModel.mNumFaces += mesh->mNumFaces;
        mModel.mNumVertices += mesh->mNumVertices;

        ReadVertices(mesh, vertices);
        ReadIndices(mesh, indices, subset);
        ReadMaterials(material);
        ReadTextures(material,mTexMgr);

        subsets.push_back(subset);
    }
```

Here we extract the mesh, and its material. Every mesh has a material identifier, which will be used to access to the materials vector of the scene loaded to retrieve its characteristics. Then, we read the vertices, indices materials and textures.

To read the vertices we extract the position, the normals and the texture coordinates.

```
void BlenderModel::ReadVertices(aiMesh * mesh,std::vector<Vertex::Basic32> & vertices)
{
        for (UINT j = 0; j < mesh->mNumVertices; j++)
        {
            Vertex::Basic32 vertex;

            vertex.Pos.x = mesh->mVertices[j].x;
            vertex.Pos.y = mesh->mVertices[j].y;
            vertex.Pos.z = mesh->mVertices[j].z;

            vertex.Normal.x = mesh->mNormals[j].x;
            vertex.Normal.y = mesh->mNormals[j].y;
            vertex.Normal.z = mesh->mNormals[j].z;

            if (mesh->HasTextureCoords(0))
            {
                vertex.Tex.x = mesh->mTextureCoords[0][j].x;
                vertex.Tex.y = mesh->mTextureCoords[0][j].y;
            }

            vertices.push_back(vertex);
        }
}
```

Reading the indices is also easy:

```
void BlenderModel::ReadIndices(aiMesh * mesh, std::vector<USHORT> & indices, MeshGeometry::Subset subset)
{
    for (UINT c = 0; c < mesh->mNumFaces; c++)
    {
        for (UINT e = 0; e < mesh->mFaces[c].mNumIndices; e++)
        {
            indices.push_back(subset.VertexStart + mesh->mFaces[c].mIndices[e]);
        }

    }
}
```

We just need to take into account that we must add the subset.vertexStart to the given index in order to differenciate the index 0 of a subset that the index 0 of another subset(this is necessary because we store all the indices in the same buffer).

Then, it comes the materials:

```cpp
void BlenderModel::ReadMaterials(aiMaterial * material)
{
    Material tempMat;
    aiColor4D color(0.0f, 0.0f, 0.0f, 0.0f);

    material->Get(AI_MATKEY_COLOR_AMBIENT, color);
    tempMat.Ambient = XMFLOAT4(color.r, color.g, color.b, color.a);

    material->Get(AI_MATKEY_COLOR_DIFFUSE, color);
    tempMat.Diffuse = XMFLOAT4(color.r, color.g, color.b, color.a);

    material->Get(AI_MATKEY_COLOR_SPECULAR, color);
    tempMat.Specular = XMFLOAT4(color.r, color.g, color.b, color.a);

    material->Get(AI_MATKEY_COLOR_REFLECTIVE, color);
    tempMat.Reflect = XMFLOAT4(color.r, color.g, color.b, color.a);

    if (tempMat.Ambient.x == 0 && tempMat.Ambient.y == 0 && tempMat.Ambient.z == 0 && tempMat.Ambient.w == 0)
        tempMat.Ambient = XMFLOAT4(0.5f, 0.5f, 0.5f, 1.0f);

    if (tempMat.Diffuse.x == 0 && tempMat.Diffuse.y == 0 && tempMat.Diffuse.z == 0 && tempMat.Diffuse.w == 0)
        tempMat.Diffuse = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);

    if (tempMat.Specular.x == 0 && tempMat.Specular.y == 0 && tempMat.Specular.z == 0 && tempMat.Specular.w == 0)
        tempMat.Specular = XMFLOAT4(0.6f, 0.6f, 0.6f, 16.0f);

    Materials.push_back(tempMat);

}
```

We retrieve the information of the material. If the material does not have a specific component. we set a default value.

Then we read the textures. We had a lot problems in this part. Apparently, the .tga textures gives problems when loading with directx so we had to convert them to bmp files and load them with the extension .bmp (that’s when replace method comes to place).

```cpp
void BlenderModel::ReadTextures(aiMaterial *material, TextureMgr * mTexMgr)
{
    aiString path;
    std::string texDirectory = "Models";
    if (material->GetTextureCount(aiTextureType_DIFFUSE) > 0 && material->GetTexture(aiTextureType_DIFFUSE, 0, &path) == AI_SUCCESS)
    {
        std::string fullPath = texDirectory + path.data;
        fullPath.replace(fullPath.length() - 3, fullPath.length(), "bmp");

        std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> stringConverter;
        std::wstring fullPathConverted = stringConverter.from_bytes(fullPath);

        ID3D11ShaderResourceView* texture = mTexMgr->CreateTexture(fullPathConverted);
        textureResourceView.push_back(texture);
    }

}
```

Once all the model is loaded, we then need to create the index and vertex buffers:

```
mModel.mSubsetCount = subsets.size();

mModel.Mesh.SetSubsetTable(subsets);
mModel.Mesh.SetIndices(md3dDevice, &indices[0], indices.size());
mModel.Mesh.SetVertices(md3dDevice, &vertices[0], vertices.size());

D3D11_BUFFER_DESC vbd;
vbd.Usage = D3D11_USAGE_IMMUTABLE;
vbd.ByteWidth = sizeof(Vertex::Basic32) * vertices.size();
vbd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vbd.CPUAccessFlags = 0;
vbd.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA vinitData;
vinitData.pSysMem = &vertices[0];

HR(md3dDevice->CreateBuffer(&vbd, &vinitData, &mModel.VertexBuffer));

D3D11_BUFFER_DESC ibd;
ibd.Usage = D3D11_USAGE_IMMUTABLE;
ibd.ByteWidth = sizeof(UINT) * indices.size();
ibd.BindFlags = D3D11_BIND_INDEX_BUFFER;
ibd.CPUAccessFlags = 0;
ibd.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA iinitData;
iinitData.pSysMem = &indices[0];

HR(md3dDevice->CreateBuffer(&ibd, &iinitData, &mModel.IndexBuffer));
```

Then all that is left is to render the model

```
void BlenderModel::Render(CXMMATRIX world)
{
    ID3DX11EffectTechnique* activeTech = Effects::BasicFX->Light0TexTech;
    md3dImmediateContext->IASetInputLayout(InputLayouts::Basic32);

    UINT Stride = sizeof(Vertex::Basic32);
    UINT Offset = 0;

    Effects::BasicFX->SetEyePosW(mCam->GetPosition());
    XMMATRIX worldInvTranspose = MathHelper::InverseTranspose(world);
    XMMATRIX worldViewProj = world * mCam->ViewProj();

    Effects::BasicFX->SetWorld(world);
    Effects::BasicFX->SetWorldInvTranspose(worldInvTranspose);
    Effects::BasicFX->SetWorldViewProj(worldViewProj);
    Effects::BasicFX->SetTexTransform(XMMatrixIdentity());

    D3DX11_TECHNIQUE_DESC techDesc;
    activeTech->GetDesc(&techDesc);
    md3dImmediateContext->RSSetState(0);
    for (UINT p = 0; p < techDesc.Passes; ++p)
    {
        md3dImmediateContext->IASetVertexBuffers(0, 1, &mModel.VertexBuffer, &Stride, &Offset);
        md3dImmediateContext->IASetIndexBuffer(mModel.IndexBuffer, DXGI_FORMAT_R32_UINT, 0);
        for (UINT i = 0; i < mModel.mSubsetCount; i++)
        {
            Effects::BasicFX->SetMaterial(Materials[i]);
            Effects::BasicFX->SetDiffuseMap(textureResourceView[i]);
            activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
            mModel.Mesh.Draw(md3dImmediateContext, i);
        }
```

Note that we select:

- First the index and vertex buffers
- Then, the material and the texture
- Lately, draw the model

Result:

3-Blending:

Category: Intermediate - Uses color blending and alpha blending

In this assignment, we will implement two different blending states. One, that uses color blending, and another that uses alpha blending, to create transparency effects. First we need to fill the descriptor for each of the blending states. For the color blending we decided to use the source color as source blending, the inverse blending factor as the destination blending and the subtraction as blending operation. It looks like this:

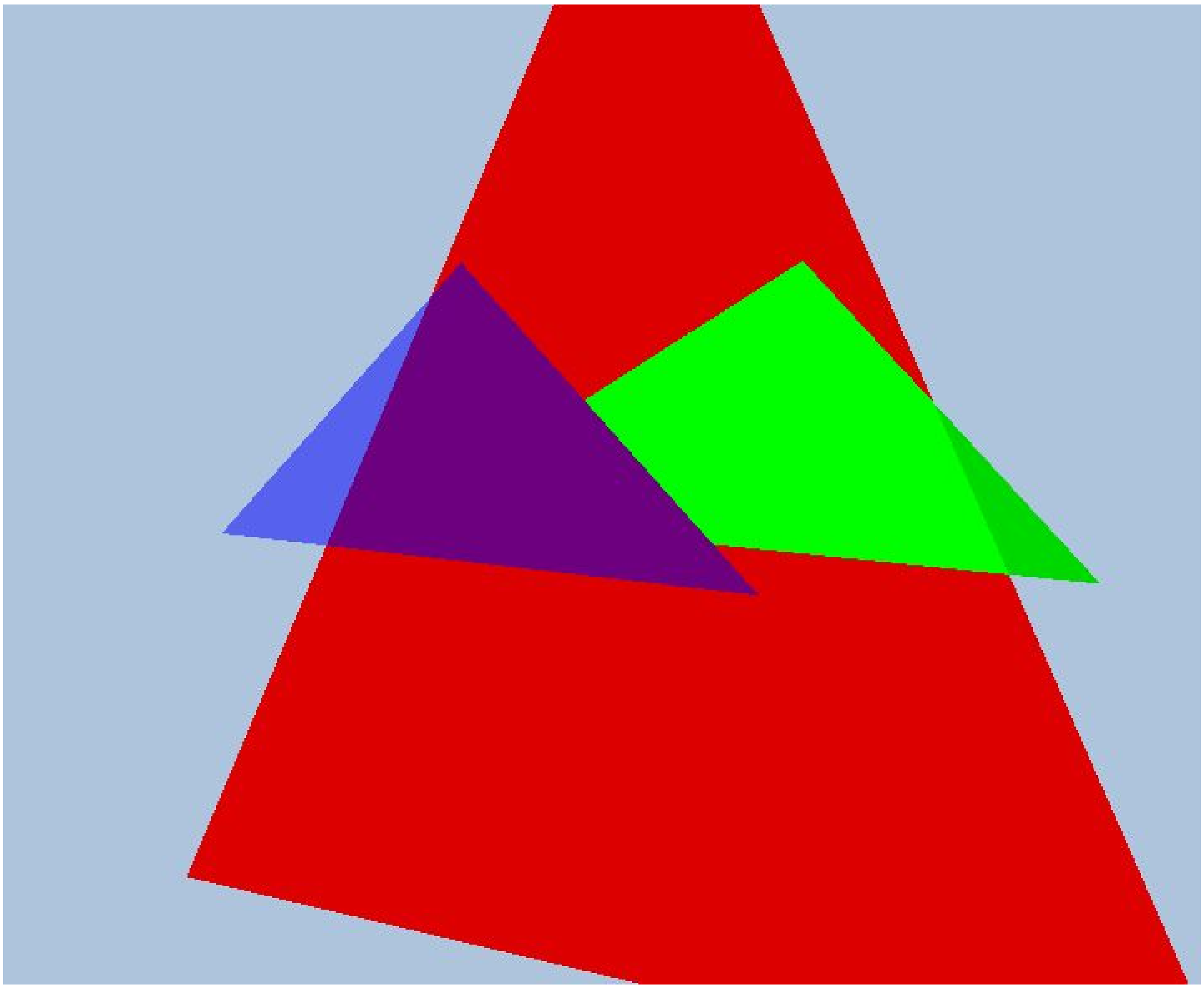
```cpp
D3D11_BLEND_DESC colorDesc = { 0 };
colorDesc.AlphaToCoverageEnable = false;
colorDesc.IndependentBlendEnable = false;

colorDesc.RenderTarget[0].BlendEnable = true;
colorDesc.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_COLOR;
colorDesc.RenderTarget[0].DestBlend = D3D11_BLEND_INV_BLEND_FACTOR;
colorDesc.RenderTarget[0].BlendOp = D3D11_BLEND_OP_SUBTRACT;
colorDesc.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
colorDesc.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
colorDesc.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;
colorDesc.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;

HR(md3dDevice->CreateBlendState(&colorDesc, &ColorBlend));
```

For the alpha blending we use the source blending we use the src alpha value and for the destination blending we use the inverse of the alpha value. For the blending operation we will use the addition. our descriptor look like this:

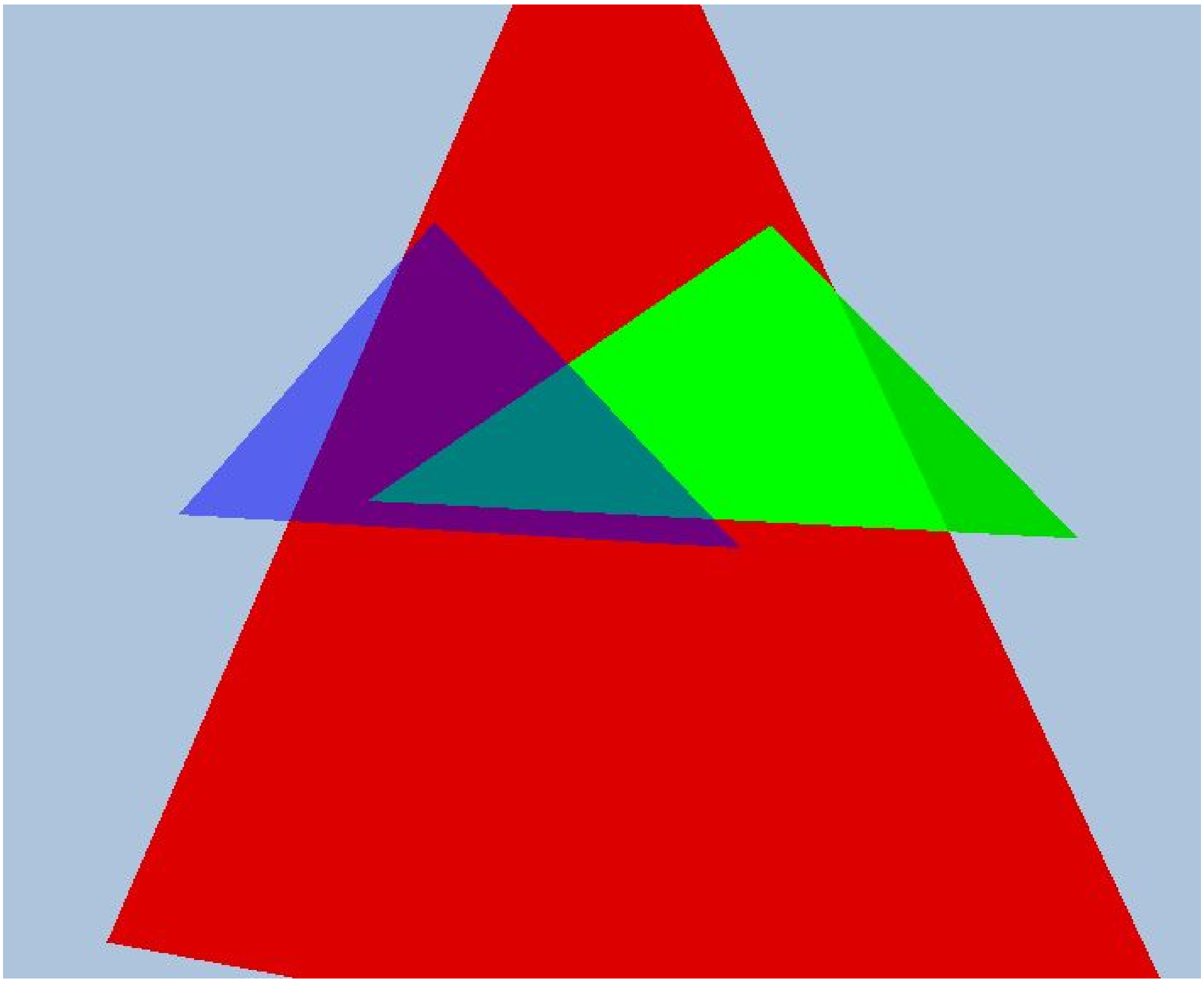
```cpp
D3D11_BLEND_DESC transparentDesc2 = { 0 };
transparentDesc2.AlphaToCoverageEnable = false;
transparentDesc2.IndependentBlendEnable = false;

transparentDesc2.RenderTarget[0].BlendEnable = true;
transparentDesc2.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_ALPHA;
transparentDesc2.RenderTarget[0].DestBlend = D3D11_BLEND_INV_SRC_ALPHA;
transparentDesc2.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
transparentDesc2.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
transparentDesc2.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
transparentDesc2.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;
transparentDesc2.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;

HR(md3dDevice->CreateBlendState(&transparentDesc2, &AlphaBlend));
```

For the different tests we altern the different order of the blending states to show different effects.

In the first case we have a bad use of the alpha blending. The alpha blending should be used once all the opaques objects have been draw, because otherwise, we have something that is only transparent to the objects that were already drawn before it. For example:

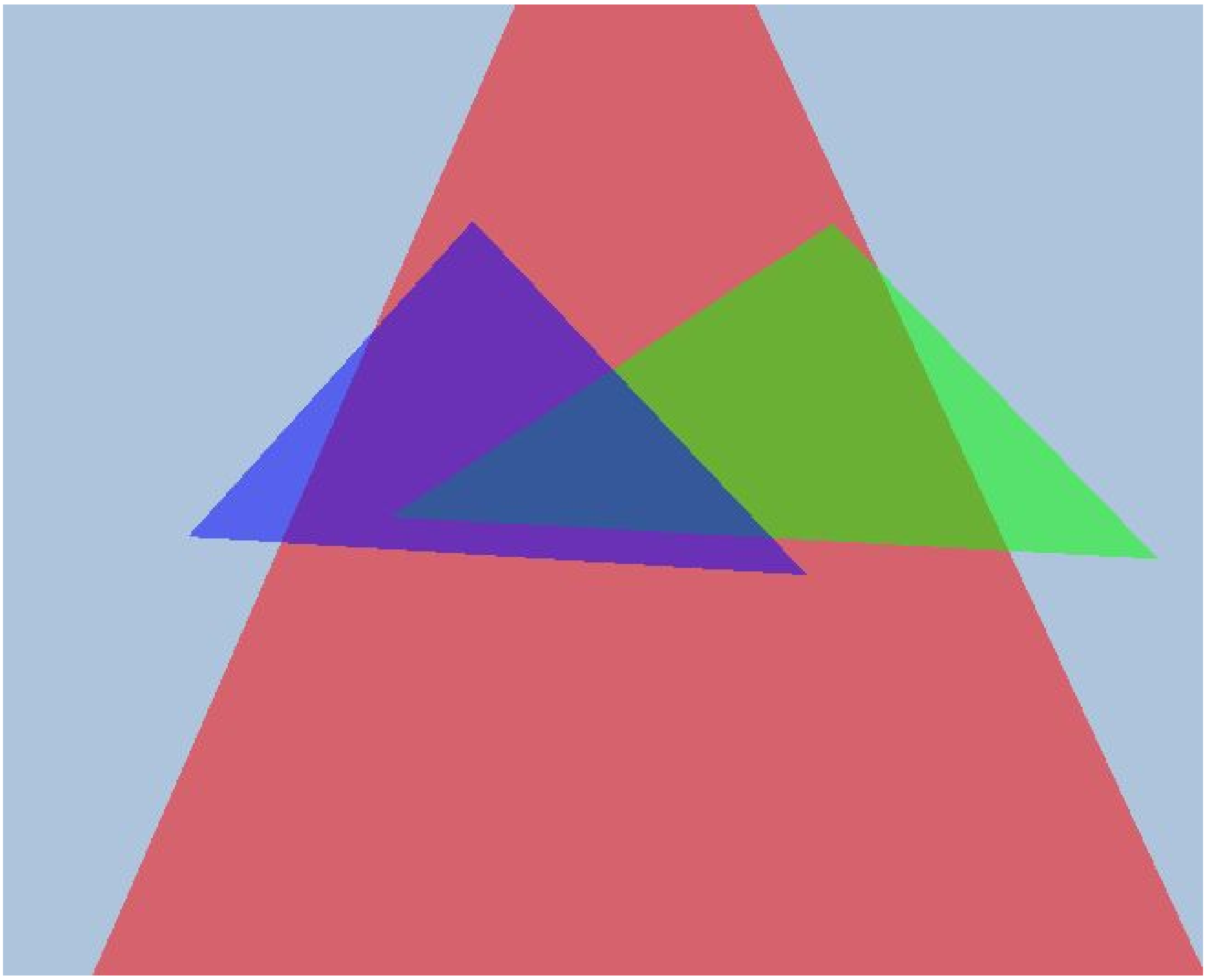


Here we rendered first the red triangle, second the blue triangle with alpha blending and third the green triangle with color blending. As you can see, the blue triangle is only transparent with the red triangle and the background(because those objects were already drawn) but not with the green. If we use the alpha blending correctly we would have the second scenario:



Here, first we draw the red, then the green and lastly, the blue one. Note that here the blue is transparent to all the objects because it is the last triangle being drawn.

For the last scenario,we have the alpha blending applied correctly to the three triangles:



4-Lighting:

Category: Advanced - Toon Shading

To implement the toon shading described in luna's book, we need to implement in the .fx file that deals with the light (LightHelper.fx) the given function(extracted from luna):

$$k'_d = f(k_d) = \begin{cases} 0.4 & \text{if} \quad -\infty < k_d \le 0.0 \\ 0.6 & \text{if} \quad 0.0 < k_d \le 0.5 \\ 1.0 & \text{if} \quad 0.5 < k_d \le 1.0 \end{cases}$$

$$k'_s = g(k_s) = \begin{cases} 0.0 & \text{if} \quad 0.0 \le k_s \le 0.1 \\ 0.5 & \text{if} \quad 0.1 < k_s \le 0.8 \\ 0.8 & \text{if} \quad 0.8 < k_s \le 1.0 \end{cases}$$

So, we implement this function in the shader language, for both, the specular and the diffuse factor:

```
float toonSpecFactor(float specFactor) {
    if (specFactor <= 0.1 && specFactor >= 0.0) {
        specFactor = 0.0;
    }
    else {
        if (specFactor <= 0.8) {
            specFactor = 0.5;
        }
        else {
            specFactor = 0.8;
        }
    }
    return specFactor;
}

float toonDiffuseFactor(float diffuseFactor) {
    if (diffuseFactor <= 0.0) {
        diffuseFactor = 0.4;
    }
    else {
        if (diffuseFactor <= 0.5) {
            diffuseFactor = 0.6;
        }
        else {
            diffuseFactor = 1.0;
        }
    }
    return diffuseFactor;
}
```

Then, once we have calculated the diffuse factor and the specular factor, we apply the functions described above to the specular factor and diffuse factor(we apply the function to both factors in the directional light, spot light and point light function):

```
float diffuseFactor = dot(lightVec, normal);
diffuseFactor = toonDiffuseFactor(diffuseFactor)

float specFactor = pow(max(dot(v, toEye), 0.0f), mat.Specular.w);
specFactor = toonSpecFactor(specFactor)
```

These functions used to calculate the different types of lights will be used within the pixel shader.

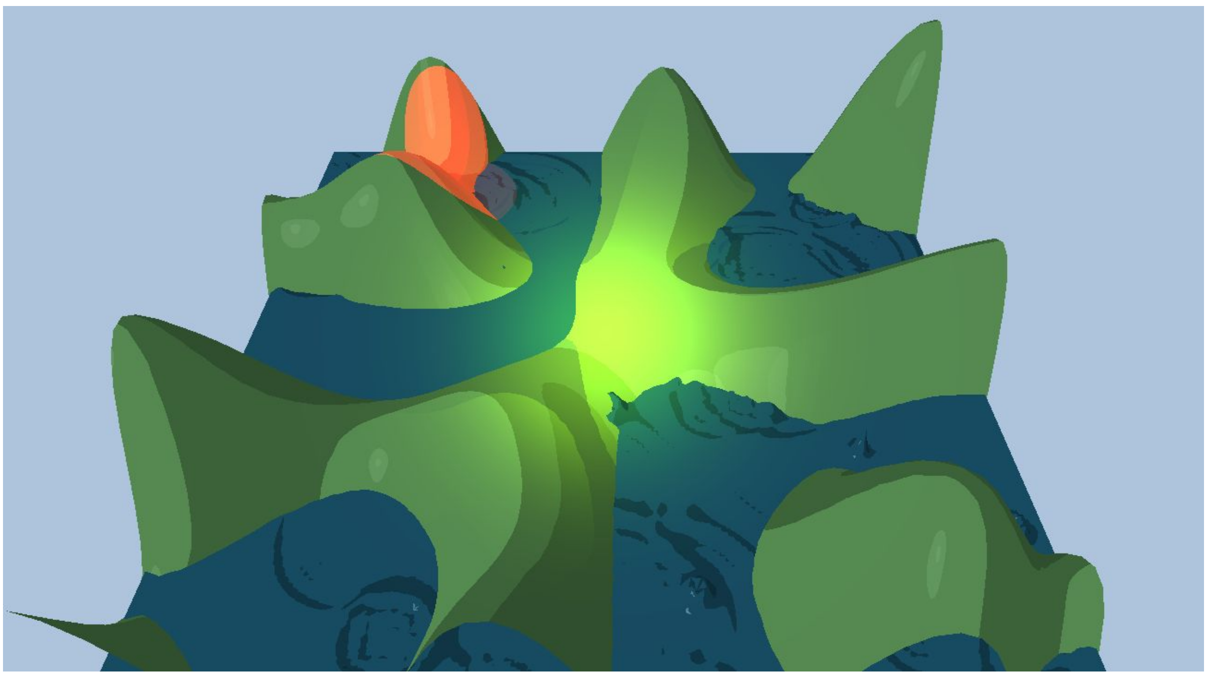
Also, we implemented some input to our project.

- 1 and 2 decreases or increases the spot light cone of light. We do so by increasing the spot component of the light.
- R, G and B changes the point light of the scene to the color red, green and blue respectively.

All the described above is implemented like this:

```
void LightingApp::UpdateScene(float dt)
{

    if (GetAsyncKeyState('1') & 0x8000)
        mSpotLight.Spot += 2.0f;

    if (GetAsyncKeyState('2') & 0x8000)
        mSpotLight.Spot -= 2.0f;

    if (GetAsyncKeyState('R') & 0x8000) {
        mPointLight.Ambient = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
        mPointLight.Diffuse = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
        mPointLight.Specular = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
    }
    if (GetAsyncKeyState('G') & 0x8000) {
        mPointLight.Ambient = XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f);
        mPointLight.Diffuse = XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f);
        mPointLight.Specular = XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f);
    }

    if (GetAsyncKeyState('B') & 0x8000) {
        mPointLight.Ambient = XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f);
        mPointLight.Diffuse = XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f);
        mPointLight.Specular = XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f);
    }
```

Result:



5-Texturing:

Category: Advanced - Mobile phone textured with the scene in its screen.

Our phone consists of 3 quads. One for the front of the phone, one for the back of the phone and another for the screen. To do so first we create the vertices in the BuildGeometryBuffers() function.

```
    //phone
    vertices[0].Pos = XMFLOAT3(-2.5, -5, -3);
    vertices[1].Pos = XMFLOAT3(-2.5,5, -3);
    vertices[2].Pos = XMFLOAT3(2.5, 5, -3);
    vertices[3].Pos = XMFLOAT3(2.5, -5, -3);

    vertices[0].Tex = XMFLOAT2(0,1);
    vertices[1].Tex = XMFLOAT2(0,0);
    vertices[2].Tex = XMFLOAT2(1,0);
    vertices[3].Tex = XMFLOAT2(1,1);

    //screen
    vertices[4].Pos = XMFLOAT3(-2.2, -3.8, -3.01);
    vertices[5].Pos = XMFLOAT3(-2.2, 4.0, -3.01);
    vertices[6].Pos = XMFLOAT3(2.2, 4.0, -3.01);
    vertices[7].Pos = XMFLOAT3(2.2, -3.8,-3.01);

    vertices[4].Tex = XMFLOAT2(0,1);
    vertices[5].Tex = XMFLOAT2(0,0);
    vertices[6].Tex = XMFLOAT2(1,0);
    vertices[7].Tex = XMFLOAT2(1,1);
```

Note that we assign manually the texture coordinates as well. We also create a box, which will be our scene to represent on the mobile's screen.The texture coordinates of the box are manually set so it will work with our dice texture.

The textures used for the phone have an alpha channel, so we will have to activate the blending in order to see them correctly. First we create a blending descriptor in the Init() function and bind it to the device. We will set this blending state before drawing the quads that have the textures.

```
    D3D11_BLEND_DESC transparentDesc = {0};
    transparentDesc.AlphaToCoverageEnable = false;
    transparentDesc.IndependentBlendEnable = false;

    transparentDesc.RenderTarget[0].BlendEnable = true;
    transparentDesc.RenderTarget[0].SrcBlend       = D3D11_BLEND_SRC_ALPHA;
    transparentDesc.RenderTarget[0].DestBlend      = D3D11_BLEND_INV_SRC_ALPHA;
    transparentDesc.RenderTarget[0].BlendOp        = D3D11_BLEND_OP_ADD;
    transparentDesc.RenderTarget[0].SrcBlendAlpha  = D3D11_BLEND_ONE;
    transparentDesc.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
    transparentDesc.RenderTarget[0].BlendOpAlpha   = D3D11_BLEND_OP_ADD;
    transparentDesc.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;

    HR(md3dDevice->CreateBlendState(&transparentDesc, &TransparentBS));
```

Then, in the Init function as well, we have to create our shader resources. Because we have our textures in a file, we import the textures for the front, the back and the box like this:

```
    HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
        L"Textures/iphone.png", 0, 0, &mDiffuseMapSRV, 0 ));

    HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
        L"Textures/dice.png", 0, 0, &mScreen, 0 ));

    HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
        L"Textures/backiphone.png", 0, 0, &mBackIphone, 0 ));
```

Now, for the screen we need to create it in a different way. First we create a descriptor for the texture

```
    D3D11_TEXTURE2D_DESC texDesc;

    texDesc.Width     = mClientWidth;
    texDesc.Height    = mClientHeight;
    texDesc.MipLevels = 1;
    texDesc.ArraySize = 1;
    texDesc.Format    = DXGI_FORMAT_R8G8B8A8_UNORM;
    texDesc.SampleDesc.Count   = 1;
    texDesc.SampleDesc.Quality = 0;
    texDesc.Usage          = D3D11_USAGE_DEFAULT;
    texDesc.BindFlags      = D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_UNORDERED_ACCESS;
    texDesc.CPUAccessFlags = 0;
    texDesc.MiscFlags      = 0;

    ID3D11Texture2D* offscreenTex = 0;
    HR(md3dDevice->CreateTexture2D(&texDesc, 0, &offscreenTex));
```

Then lately, we create the shader resource and a new render target view.

```
    HR(md3dDevice->CreateShaderResourceView(offscreenTex, 0, &mOffscreenSRV));
    HR(md3dDevice->CreateRenderTargetView(offscreenTex, 0, &mOffscreenRTV));
```

Once that all is created, we just need to go to the DrawScene() function.

```
void CrateApp::DrawScene()
{
    //Set the render target to the texture's render target
    ID3D11RenderTargetView* renderTargets[1] = {mOffscreenRTV};
    md3dImmediateContext->OMSetRenderTargets(1, renderTargets, mDepthStencilView);

    md3dImmediateContext->ClearRenderTargetView(mOffscreenRTV, reinterpret_cast<const float*>(&Colors::Silver));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView, D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);

    //Draw scene as usual.
    DrawSceneNormal();

    //Switch to the regular render target
    renderTargets[0] = mRenderTargetView;
    md3dImmediateContext->OMSetRenderTargets(1, renderTargets, mDepthStencilView);

    md3dImmediateContext->ClearRenderTargetView(mRenderTargetView, reinterpret_cast<const float*>(&Colors::Silver));
    md3dImmediateContext->ClearDepthStencilView(mDepthStencilView, D3D11_CLEAR_DEPTH|D3D11_CLEAR_STENCIL, 1.0f, 0);

    //Draw the scene again
    DrawScreen();

    HR(mSwapChain->Present(0, 0));
}
```

In order to render the scene to the screen we do the following:

- We change to the offScreen render target we previously defined. The scene will be rendered there and we won't see it.
- We draw the scene as usual from the point of view of the phone.*
- Switch to the normal render target and draw the scene again.

*Actually, the camera is situated farther of the phone in order to look nicer.

To draw the scene to the offscreen render target, we use a different point of view which will be used to render the scene. We do so defining a new view matrix in the constructor.

```
XMVECTOR pos = XMVectorSet(0, 0, -7, 1.0f);
XMVECTOR target = XMVectorZero();
XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

XMMATRIX V = XMMatrixLookAtLH(pos, target, up);
XMStoreFloat4x4(&mCameraView, V);
```

And then, using it in the DrawSceneNormal() function:

```
XMMATRIX view = XMLoadFloat4x4(&mCameraView);
```

Doing so, the scene will be rendered from the position we defined and not the position that the normal camera is.

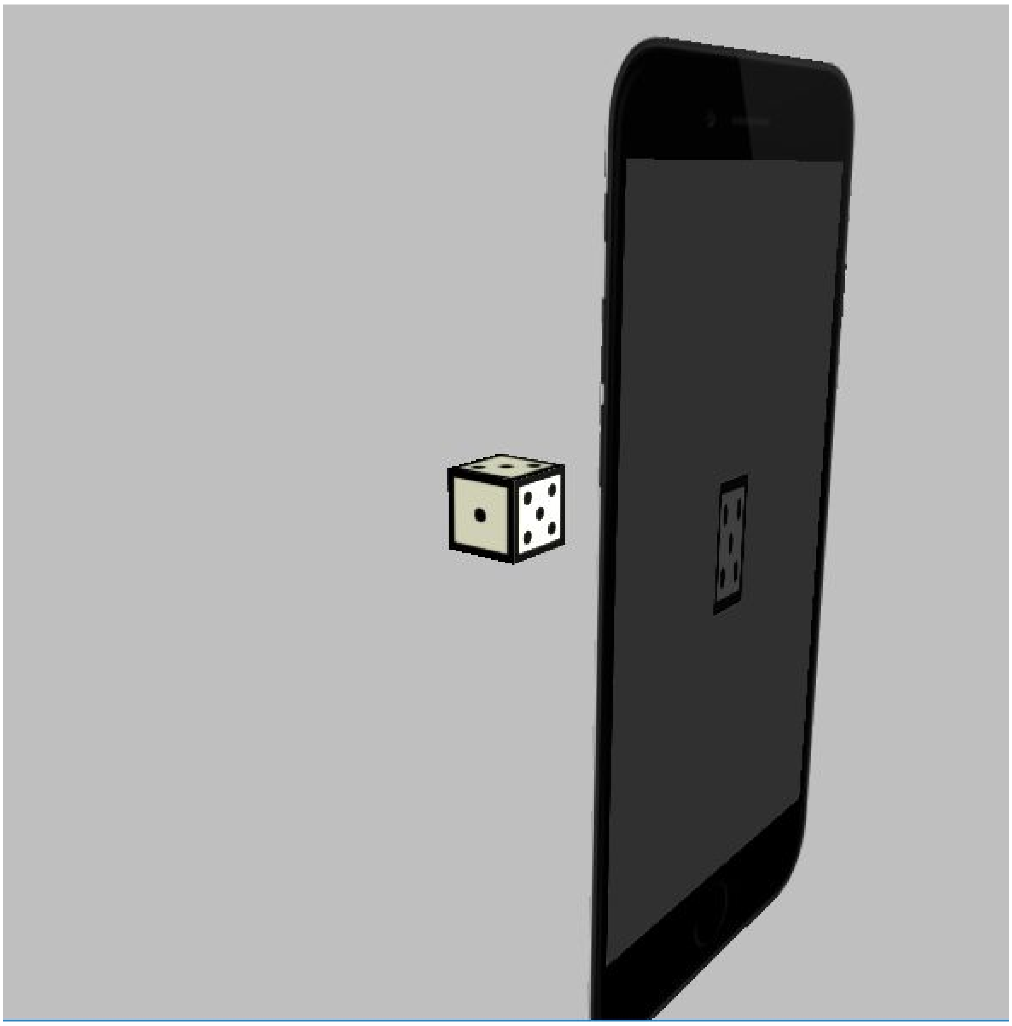
In this method we only draw the box for convenience. Because the camera is situated farther than the phone, drawing also the phone will result in a phone in the way of the box so we wouldn't be able to see it.

After drawing the scene to the offscreen render target, we render the scene to the regular render target.

```
//Front phone
Effects::BasicFX->SetDiffuseMap(mDiffuseMapSRV);
md3dImmediateContext->RSSetState(NoCullRS);
md3dImmediateContext->OMSetBlendState(TransparentBS, blendFactor, 0xffffffff);
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(6, 0, 0);

//Screen
Effects::BasicFX->SetDiffuseMap(mOffscreenSRV);
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(6, 0, 4);

//Back phone
Effects::BasicFX->SetDiffuseMap(mBackIphone);
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(6, 12, 8);

//Box
Effects::BasicFX->SetDiffuseMap(mScreen);
activeTech->GetPassByIndex(p)->Apply(0, md3dImmediateContext);
md3dImmediateContext->DrawIndexed(mBoxIndexCount, mBoxIndexOffset, mBoxVertexOffset);
```

Note that, we activate the blend state we defined previously so the textures will be displayed with the alpha channel and we will be able to see transparency. For the screen we set the diffuse map to the offscreen texture we defined earlier. That texture will hold the previous the scene previously rendered.

Result:



6-Shading:

Category: Advanced - Shader of high complexity (200 LOC)

The shader implemented is this: https://www.shadertoy.com/view/Mss3WN
In order to port it we had to do these things:

- Change the type vec2, vec3, vec4, mat3 to its type in HLSL language (float2 float3 float4 and float3x3).
- Change the function fract(..) to frac(...).
- Change the function mix(..), linear interpolation to lerp(...).
- Make the array of balls that are moving (global variable) static, because we change its value during execution (otherwise it won't compile).
- Change the multiplicator operator * by mul(..) when multiplying vector with matrices
- The shader makes use of resolution and the mouse position to display the balls. We set these values to constant values so we don't have to bind the values of the app to the shader.

- In order to being animate, the shader needs to keep track of the time with the variable iGlobalTime. So we have to link the time passed within the application to the shader, to do so we had to follow these steps:

1- Define a new variable in the shader called iGlobalTime:

```
cbuffer globalTime
{
    float iGlobalTime;
};
```

2-When building the fx file to the app, bind the iGlobalTime, to a variable in the c++ application, this is done by adding this line to the BuildFX() function:

```
mGlobalTime = mFX->GetVariableByName("iGlobalTime")->AsScalar();
```

3- Then, in order to update its value, we set the value in the updatScene() to the timer's value of the application:

```
mGlobalTime->SetFloat(mTimer.TotalTime());
```

Then, lately, we copy the code of the mainImage(...) method(shadertoy) within our pixel shader, and assign the the resulting color calculated, to the pixel's color.

Result: