Writing a 3D DirectX maze that's navigable with speech commands

This article covers the creation of a 3D Direct3D rendered maze for Windows Phone 8 that is navigable with voice commands

Introduction

The article covers the creation of a 3D Direct3D rendered maze for Windows Phone 8 that is navigable with voice commands.

The application starts with player's avatar on the black starting tile. User uses voice commands ("up", "down", "left" and "right") to guide the avatar through the maze to the white finish tile which then results in avatar reseting back to start for another try.

Screenshots

A couple of screenshots of the application in action:

Application listening for voice input

A full render between inputs

What's new in Windows Phone 8

There are a lot of new things introduced with the arrival of Windows Phone 8. We'll quickly go through the ones relevant to this article.

The most important thing for game developers introduced with WP8 is the ability to use native code. Even though this means that applications using this feature will not be backwards compatible with WP7 devices, this one is a welcome change.

Native code brings about the power and performance of C++ and the ability to use Microsoft's acclaimed DirectX directly (as compared to XNA framework). Developers get the power of the Direct3D (3D Rendering API; a subset of DirectX – the latter also includes APIs for audio, input, etc.) with fully programmable shaders (a huge step from WP7 which had moreless a fixed shader set). The shaders are required to be precompiled but most scenarios have no need for shader compilation on-the-fly. Since most of graphical intensive work is, and should be done on the GPU (which uses shader instructions to carry out the tasks), programmable shaders make a welcome introduction.

C++ also means the logic part of the code is portable among a wide variety of platforms: there's only the need for differentiation on the system-close code.

Another welcome thing is that Microsoft is converging all of it's platforms under the same codebase, so the same code can be reused on WP8, Windows 8 Desktops, tablets, etc.

The Voice\Speech control API also get a big update - developers will be able to use it in their applications from now on. And not just in the WP7 scenario, where voice commands were used to launch the application. Here developer can use this functionality from within his own app.

Starting up

(C) Copyright Nokia 2013. All rights reserved.

We will take the Direct3D App template as our starting point. This provides us with a common ground to build upon, as the very basics of Direct3D have a steep learning curve. It takes a lot of work just to display a single triangle on the screen and that to some extent is where the template drops us off. It is very good to know the lower level, especially once you start delving deeper into the graphical engine, but this article starts off a bit further to cover more real world usage topics. If you want a comprehensive look at the basics I suggest visiting http://www.directxtutorial.com/.

Direct3D App template

The template provides us with a nice rich standpoint.

<projectname>.h & .cpp includes app lifetime event handlers and a very nice game loop with Render, Update and Present methods. Template also includes Direct3DBase.h & .cpp which are responsible for all low level initialization: creating devices, drawing surfaces, buffer swap chain, etc.

Lastly we have CubeRenderer.h & .cpp files which will be our main point of interest. It is the place where we will start with our changes which first being rename to `CGame`. It already has functionality for vertex data loading, drawing the vertex data with index buffers, transformations, a take on the virtual game camera and both types of shaders needed to display geometry on screen.

Shaders are coded in High Level Shading Language or HLSL. And they must be precompiled once the app ships. This is done automatically inside the visual studio.

Additions

We will add textures, basic lighting, mesh loading, basic shadowing in terms of appearance and rendering . For input we will implement the newly accessible layer of voice control for maze navigation.

New Vertex Data structure

We will be using vertex and index buffers too with slight changes. The vertex data in the template includes a `Color` parameter which we won't be needing, because all the faces we will draw will have textures on them.

We'll change this:

```
struct ModelViewProjectionConstantBuffer
{
 DirectX::XMFLOAT4X4 model;
 DirectX::XMFLOAT4X4 view;
 DirectX::XMFLOAT4X4 projection;
};

struct VertexPositionColor
{
 DirectX::XMFLOAT3 pos;
 DirectX::XMFLOAT3 color;
};

const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
 { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,  D3D11_INPUT_PER_VERTEX_DATA, 0 },
 { "COLOR",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

To this:

```
struct PerFrameCB
{
 DirectX::XMFLOAT4X4 view;
 DirectX::XMFLOAT4X4 projection;

    DirectX::XMFLOAT4 lightVector;
    DirectX::XMFLOAT4 lightColor;
    DirectX::XMFLOAT4 ambientColor;
};

struct PerObjectCB
{
 DirectX::XMFLOAT4X4 modeltransform;
};

struct VertexPosNormTex
{
 DirectX::XMFLOAT4 pos;
 DirectX::XMFLOAT4 normal;
 DirectX::XMFLOAT2 texcoord;
}

const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
 { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,  D3D11_INPUT_PER_VERTEX_DATA, 0 },
 { "NORMAL",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 16, D3D11_INPUT_PER_VERTEX_DATA, 0 },
 { "TEXCOORD",    0, DXGI_FORMAT_R32G32_FLOAT, 0, 32, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};

//and their appropriate counterparts inside the shaders
```


(C) Copyright Nokia 2013. All rights reserved.

As you can see we are changing up the structure in a few ways. First off we will have two Constant Buffers instead of one: one will be updated once per every frame being rendered and the other once per every object inside that frame.

Also we are adding two new parameters to the `D3D11_INPUT_ELEMENT_DESC`. The second one is Normals component which will be discussed in more detail in Lighting section of this article. The third one is texture coordinate component which will tell Direct3D how the texture is to be mapped to the surface.

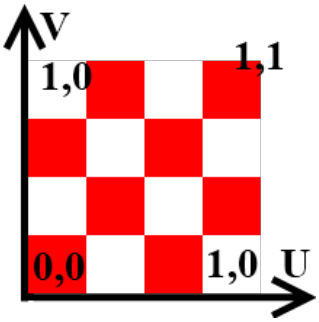
Notice some additional parameters related to lighting in the first Constant buffer too.

One disadvantage is that this means that more vertices will be needed to carry the information over. For example the template's cube uses 8 vertices to render all 12 triangle faces. In our case a vertex can only host one normal coordinate, so for every face that is rendered using a vertex in the exact same post we will be needing a new one for it to display correctly. So in our case we will need 24 vertices to render a single cube.

Textures

First off we will be adding textures. Textures add more life to rendered scene as you can go only so far with just color coded vertices as in the template.

Texture is an image that is applied to the surface of a shape. Every vertex in a polygon is assigned a texture coordinate (which is also known as a UV coordinate). So, to build one side of a box you would have 4 vertices that contain UV coordinates of a particular texture. Youd arrange those vertices in a certain order to produce two triangles that make the final face of a box. UV coordinates are a bit special. As the image that is loaded in memory is in screen coordinates (i.e. Pt 0,0 is top-left), UV coordinates range from 0 to 1 (float values) and start in the lower left. Please see the image below for illustration.



Now to load the textures into the memory and apply and render them.

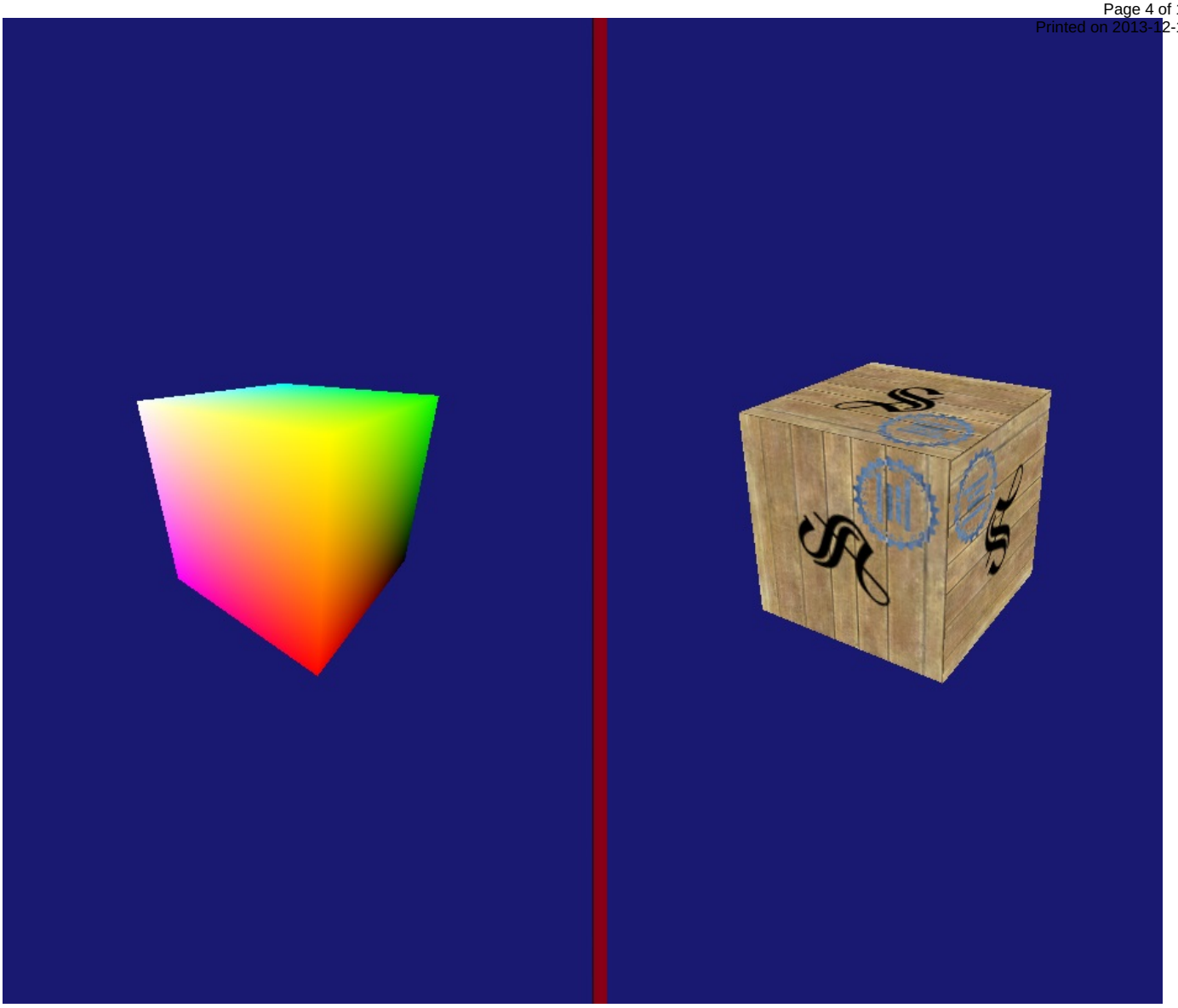
The D3Dx utility library was deprecated with windows 8 SDK. So any calls that used to begin with D3DX prefix that were math related were migrated to "DirectXmath.h", others to other appropriate places. As for texture loading, it was dismissed completely. To load in the texture we use an external library known as DirectX Tool kit. You can download it from http://directxtk.codeplex.com/. We'll use DDSTextureLoader.h & .cpp, **PlatformHelpers.h** and **dds.h** from it to load in the texture to the video memory.

The following snippet shows the creation, seting the desired texture for the object to have it's faces textured and the draw call.

```
CreateDDSTextureFromFile(aDev.Get(), L"Assets\\Textures\\avatar.dds", NULL, m_texture.GetAddressOf()) ;//create texture: using id3d
....
VertexPosNormTex OurVertices[] =
   {
VertexPosNormTex(Vector3D(-inSize.X/2, -inSize.Y/2, -inSize.Z/2), Vector3D(-1.0f, 0.0f, 0.0f), Vector3D(0.0f, 0.0f, 0.0f)),//the la
   VertexPosNormTex(Vector3D(-inSize.X/2, -inSize.Y/2, inSize.Z/2), Vector3D(-1.0f, 0.0f, 0.0f), Vector3D(0.0f, 1.0f, 0.0f)),
   VertexPosNormTex(Vector3D(-inSize.X/2, inSize.Y/2, -inSize.Z/2), Vector3D(-1.0f, 0.0f, 0.0f), Vector3D(1.0f, 0.0f, 0.0f)),
   VertexPosNormTex(Vector3D(-inSize.X/2, inSize.Y/2, inSize.Z/2), Vector3D(-1.0f, 0.0f, 0.0f), Vector3D(1.0f, 1.0f, 0.0f)),//note
};
...
aDevCon->PSSetShaderResources(0, 1, m_texture.GetAddressOf()); //then we set the texture to be applied as the pixel shader resource
...
//setting the vertex and index buffers to draw from before the final draw call
aDevCon->DrawIndexed(m_IndexCount, 0, 0);//the final draw call
```

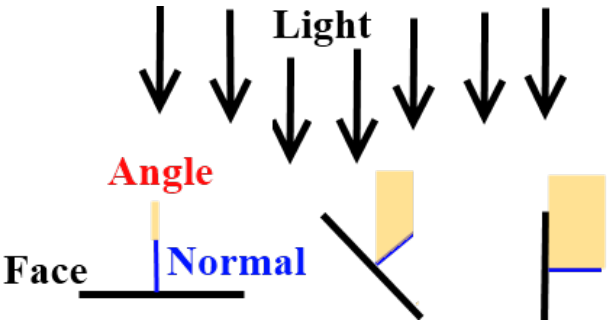
The textures used are of DDS (or DirectDraw Surface) file format. It's a compressed image format that can be decompressed in hardware by the GPU. It's possible to write your own texture loader which would allow to directly load in other more popular picture formats, but that would be quite complex as it would require decompression and making the data ready for the gpu. Even though most of the free painting applications have no support for this file format, there are quite a few converters out there like http://www.imageconverterplus.com/ or http://www.easy2convert.com/png2dds/ to name a few.

 (C) Copyright Nokia 2013. All rights reserved.



Basic Lighting

Next is the lighting. You have seen the light workings foreshadowed in Vertex structure. When we talk about lighting in 3D graphics, we talk about normals. A normal is a Vector that is perpendicular to the face that the light falls onto. As the surface tilts farther away from the light, the normal vector becomes more exposed to the light as it is perpendicular. Direct3D compares the angle of the normal to the angle of the light to determine how brightly lit the surface should be. The surface is brighter the closer these angles are. See the image below.



We illuminate everything using the ambient lighting. The light direction, and colour is provided to the shader via the constant buffer which is a means to provide shader with desired data.

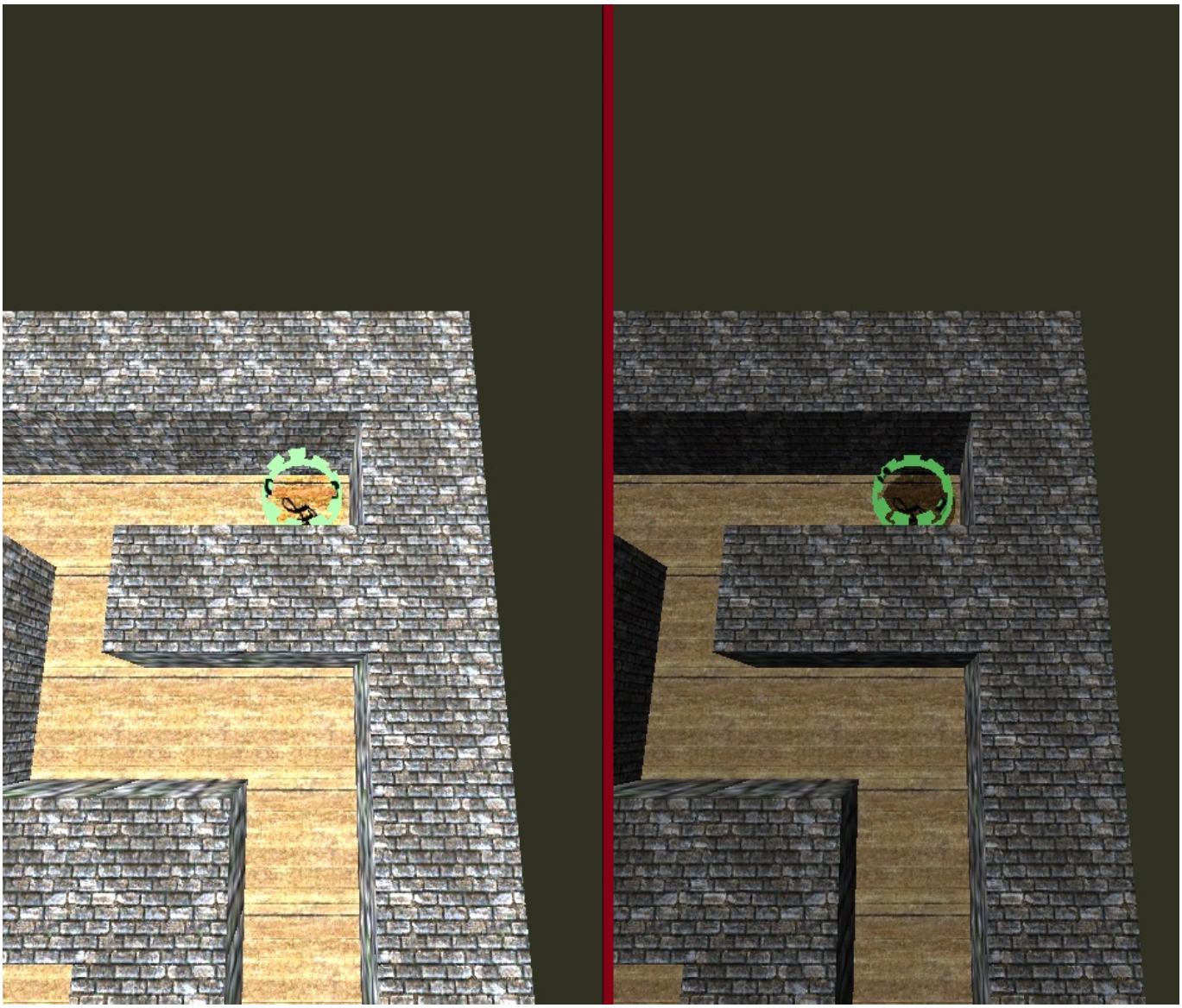
```
cbuffer PerFrameCB : register(b0)
{
 matrix view;
 matrix projection;

 float4 lightVector;
 float4 lightColor;
 float4 ambientColor;
};
...

output.color = ambientColor;//basic color

float diffusebrightness = saturate(dot(input.normal, lightVector));//surface brightness calculation
   output.color += lightColor * diffusebrightness;//adding the brightness and the color of the light to the final color
```

 (C) Copyright Nokia 2013. All rights reserved.



Mesh loading

Coding in every single vertex, their normal data and texture coordinates can get a bit tedious. Now imagine you want to construct something like a car model or a human avatar. They might have hundreds if not tens thousands of vertices. Populating this much data by hand would take too much time and is just an overhaul. And the game artists are usually lazy enough to not know coding at all. That's where mesh loading comes in.

It's a lot easier to model something in an external 3D modelling application like 3D Studio Max or Blender and then export it as a mesh. Then all is left is loading it into your app.

In our case we use .OBJ file format which is very easy to parse, is supported by most 3D modelling applications and has the ability to store vertex positions, normals, indexes and texture coordinates - all that we need. You can read more about obj file format here: http://en.wikipedia.org/wiki/Wavefront_.obj_file

The way it is loaded is: we parse the file and create Vertex and index buffers, populate them with data and load them in for rendering as if we had them hard-coded as say cube vertex and index buffers.

The only object that uses mesh loading in this example is the player avatar. That is sufficient enough to show the idea.

Basic shadowing

Shadows are a crucial part in creating realistic looking scenes. If a scene seems somewhat unrealistic, and you can't seem to tell the relational positioning of objects to each other, most of the time it is the lack of shadows. The tricky part is that shadows are a very complex topic that should be discussed separately. It falls in advanced rendering and would increase the complexity of this article severely. That is why we will use faux shadowing instead. It's quick to render, easy to understand and uses a technique that must be discussed as it's a core element in effect building. If you want to read more on real shadowing, you can try searching for "Shadow maps". An article to get you started: http://takinginitiative.net/2011/05/15/directx10-tutorial-10-shadow-mapping/

The way we are going to achieve the faux shadowing is we'll create a simple 2D plane just below our player avatar. We will add a somewhat darkish texture to it and make it transparent. This will in turn make pixels that are obstructed by this plane darker giving an illusion of a shadow.

 (C) Copyright Nokia 2013. All rights reserved.

The technique used here is known as blending. This gives the control of how new pixels are drawn in relation to pixels already present. Think of them as local filters that change the way colours look once applied. You can increase brightness using addition (for fire or explosion effects) with every repeated draw, make pixels darker using subtraction (as in our case or maybe for smoke particle effect), filter out specific parts or colours using other options in blending operation.

```
//the blending operation descriptor structure
D3D11_BLEND_DESC bd;
bd.RenderTarget[0].BlendEnable = TRUE;
bd.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_COLOR;//the part to be taken from the present pixel for the operation
bd.RenderTarget[0].DestBlend = D3D11_BLEND_DEST_COLOR;//the part to be taken from the pixel to be drawn for the operation
bd.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;//blending operation itself
bd.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
bd.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ONE;
bd.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;//blending operation on alpha channel
bd.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;
bd.IndependentBlendEnable = FALSE;
bd.AlphaToCoverageEnable = TRUE;
...
//you might want to save your current blend state and restore it after the draw
ID3D11BlendState *savedBlendState;
aDevCon->OMGetBlendState(&savedBlendState, 0, 0);

aDevCon->OMSetBlendState(m_BlendState_ALPHA, 0, 0xffffffff);//we set the blending state to our descriptor just before the draw call
...
//restoring the previous saved blending descriptor
aDevCon->OMSetBlendState(savedBlendState, 0, 0xffffffff);
```

The only thing left is getting the right angle to correspond with ambient lighting. You can hard-code it or use the light vector's direction to calculate the offset.

This method of shadowing, albeit being very cheap in processing time and having little to no complexity in implementation is very limited. It is hard to get it display the actual geometry of the caster, it is hard to use it to self shadow an object, it can create unwanted artifacts pretty easily. It's only suitable for small applications not requiring photo-realistic imagery.

Camera View

One Last thing that is immensely useful in 3D programming is the use of matrices to generate the final view. We use View and a projection matrixes. Together they compose what approximates a virtual camera.

Using them one can achieve a moving point of view that looks at certain point, has a field of view and conforms to the aspect ratio as well as limits near and far objects for drawing.

Template uses this technique already, but it is often better to have a dedicated class for the camera, so you can easily move it around, make it snap and follow certain objects make transition effects, etc.

We will have a `CCamera` class that is pretty much the template's functionality wrapped inside a class so it's easier to use.

Input

To make any graphical application interactive there needs to be some sort of input mechanism. In our case we will be using voice commands.

Speech Recognition

We will use a set of four speech commands that the device will be listening to and once the command is hit the avatar will be moved appropriately.

Speech recognition requires a live network connection to work.

> **Note:** If you cannot get your emulator to access the network (and it even prevents Windows from accessing the Internet), chances are that you need your connection coming in through an external router that has DHCP turned on. If you have your connection coming in directly from Fibre converter or modem, chances are your emulator will not be getting the network access.

Speech recognition can be set to work in various modes:

- Free text mode - whereas it pings a remote service with the speech data and converts it into a text. This one is useful for writing apps that you can dictate large amounts of text to.
- It can be loaded with SRGS file for a more complex speech listening with various options
- small set of command words that you pass on to speech recognizer. This is the one we will be using.

We start by creating an instance of the recognizer itself. Then we pass needed commands in an array of strings. We also can change various settings for the UI.

To start the recognition process we call `Start()`. This starts the asynchronous process which we write a handler for. The handler in turn calls our handling function which extracts the captured data and creates input for the game.

```
// constructing and populating speech recogniser with data
CInput::CInput()
{
 Platform::Collections::Vector<Platform::String^> ^directions = ref new Platform::Collections::Vector<Platform::String^>;//array of
 directions->Append("up");
 directions->Append("down");
 directions->Append("left");
 directions->Append("right");

 Platform::String ^aname = {"Directions"};//just a name
```

 (C) Copyright Nokia 2013. All rights reserved.

```
 SpeechGrammarSet ^grammarSet = iSpeechUI.Recognizer->Grammars;
 grammarSet->AddGrammarFromList( aname, directions);//setting in the commands list

 m_cancelOperation = false;
 m_lastInput = (EInputCommand)-1;
 iSpeechUI.Settings->ShowConfirmation = false;//don't show what the device heard on the screen
 iSpeechUI.Settings->ReadoutEnabled = false;//prevent device from reading out loud the data on screen
}


void CInput::Start()
{

 //starting the asynchronous listening
 Windows::Foundation::IAsyncOperation<SpeechRecognitionUIResult^>^ asOp = iSpeechUI.RecognizeWithUIAsync();

 //hooking up a callback
 asOp->Completed = ref new Windows::Foundation::AsyncOperationCompletedHandler<SpeechRecognitionUIResult^>(
  [this]
 (Windows::Foundation::IAsyncOperation<SpeechRecognitionUIResult^> ^ op, Windows::Foundation::AsyncStatus status)
 {
  completedCallback(op);
  op->Cancel();

 });

}

//the callback which processes the data that was registered
void CInput::completedCallback(Windows::Foundation::IAsyncOperation<SpeechRecognitionUIResult^> ^ op)
{
 SpeechRecognitionUIResult^ res =op->GetResults();
 if (res->ResultStatus == SpeechRecognitionUIStatus::Succeeded)//if not "Sorry, didn't catch that"
 {
  Platform::String ^resText = res->RecognitionResult->Text;
  if (resText->Equals("up"))
   m_lastInput = EINPUT_UP;
  else if (resText->Equals("down"))
   m_lastInput = EINPUT_DOWN;
  else if (resText->Equals("left"))
   m_lastInput = EINPUT_LEFT;
  else if (resText->Equals("right"))
   m_lastInput = EINPUT_RIGHT;
 }
   if (!m_cancelOperation)
    Start();//start listening again
}
```

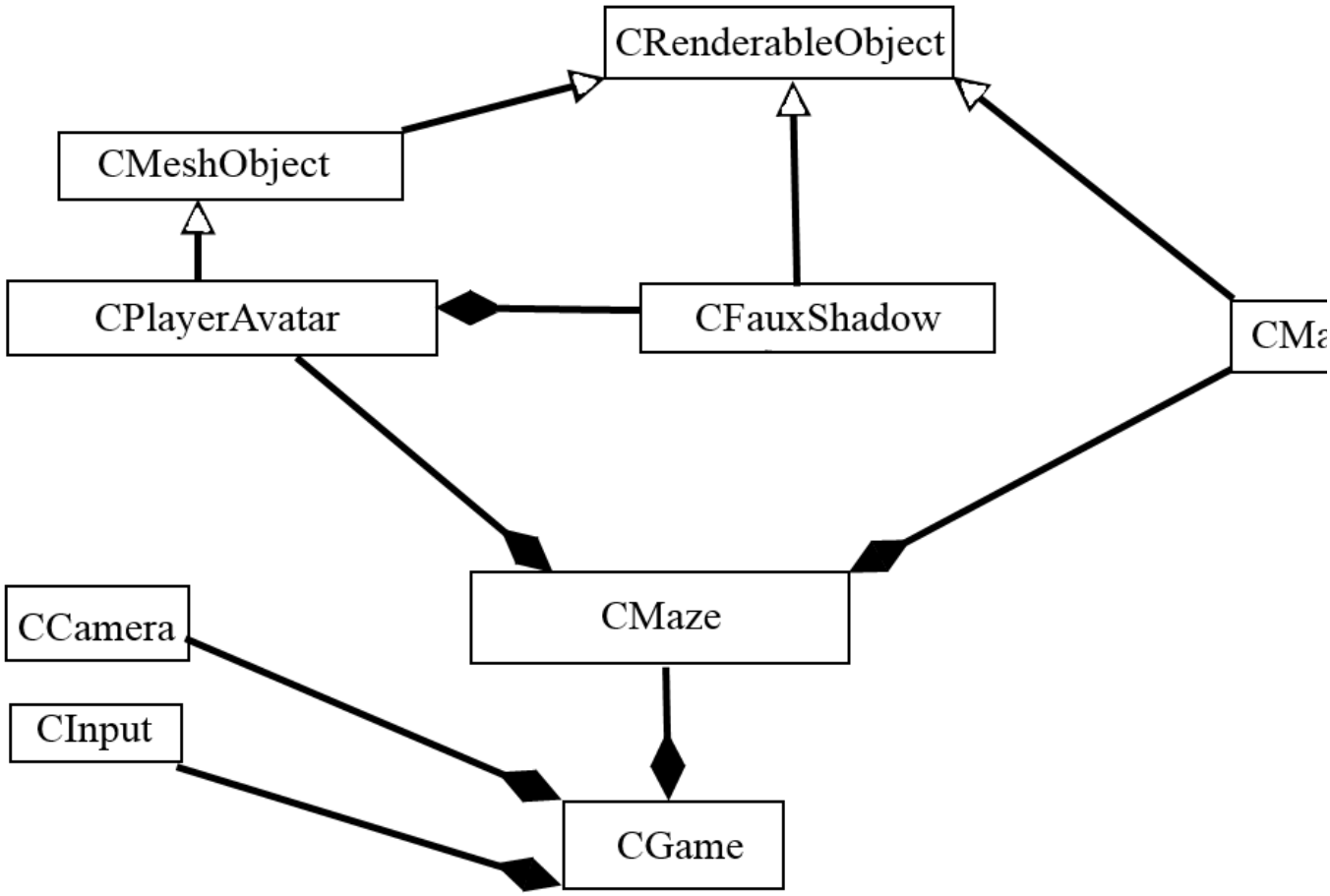
This shows how to access Speech recognition functionality from the native code. For an in depth look with various voice commands usage examples please refer to Voice_Commands_in_Windows_Phone_8

You can see the code is quite different from the rest of it. It uses WinRT which is a bit painful to use at the moment as it breaks code structure in a few ways, but its the quickest way to demonstrate the wanted capabilities of the recognizer.

Maze Workings

The structure of the example is as shown below:

 (C) Copyright Nokia 2013. All rights reserved.



It has `CRenderableObject` which has the base functionality for transformations, texture drawing and vertex & index buffer creation.

CMesh Object builds on top of that with the ability to load in an external mesh into the buffers. `CPlayerAvatar` is the class that inherits this functionality and draws the player avatar. It also has an instance of Faux shadow, so it we can simulate the shadow dropped on the ground.

We accomplish the looks of the maze using a 2 dimensional array of `CMazeElement` (which also inherits `CRenderableObject` functionality). The maze is a grid that represents this array made up of cubes with different textures for walls, path and start and finish tiles.

The maze data is initialized using a hard-coded array which is checked for enclosement before being created so not to let avatar wander off the maze.

```
//example initialization
 int KMazeData[] =
 {
    -1,0,0,0,0,0,
  1,1,0,1,1,0,
  0,0,0,1,0,1,
  0,1,1,0,0,0,
  0,1,0,0,1,0,
  0,0,0,0,1,-2
 };
 int numCols = 6;
 int numRows = 6;
```

Every single time the Render is called we redraw all the maze, an avatar and the shadow.

A rough rendering sequence:

```
  //clearing view to a specified color
  const float midnightBlue[] = { 0.198f, 0.198f, 0.139f, 1.000f };
  m_d3dContext->ClearRenderTargetView(
   m_renderTargetView.Get(),
   midnightBlue
   );
...
  //clearing depth stencil
  m_d3dContext->ClearDepthStencilView(
   m_depthStencilView.Get(),
   D3D11_CLEAR_DEPTH,
   1.0f,
   0
   );
...
   //sending data to constant buffer
   m_d3dContext->VSSetConstantBuffers(
   0,
   2,
   m_constantBuffer[0].GetAddressOf()
   );
...
      // select and bind vertex buffer to bind for drawing
         UINT stride = sizeof(VertexPosNormTex);
         UINT offset = 0;
   aDevCon->IASetVertexBuffers(0, 1, m_VBuffer.GetAddressOf(), &stride, &offset);
```

 (C) Copyright Nokia 2013. All rights reserved.

```
//binding index buffer
aDevCon->IASetIndexBuffer(m_IBuffer.Get(), DXGI_FORMAT_R16_UINT, 0);

      // select which primtive type we are using
      aDevCon->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

aDevCon->PSSetShaderResources(0, 1, m_Texture.GetAddressOf());//bind the texture


//set up transformations from object's model space
PerObjectCB cBufferObj;
XMMATRIX matRotateX, matRotateY, matRotateZ,
 matTranslate, matScale;

matRotateX = XMMatrixRotationX(m_Rotation.X);
matRotateY = XMMatrixRotationY(m_Rotation.Y);
matRotateZ = XMMatrixRotationZ(m_Rotation.Z);

matTranslate = XMMatrixTranslation(m_Position.X, m_Position.Y, m_Position.Z);

matScale = XMMatrixScaling(m_Scale.X, m_Scale.Y, m_Scale.Z);


//combine transformations
XMStoreFloat4x4(&cBufferObj.modeltransform, XMMatrixTranspose( matScale * matRotateX * matRotateY * matRotateZ*matTranslate) );

//update per object constant buffer with object's transformations
aDevCon->UpdateSubresource(aConstBuff.Get(), 0, 0, &cBufferObj, 0, 0);

      //draw the Object
      aDevCon->DrawIndexed(m_IndexCount, 0, 0);
```

When input arrives from speech recognizer, we compare the direction the player intends to go to the maze data, so to constrain avatar from leaving the area. If the player reaches the finish he is transported back to the starting tile.

Considerations

- In a real game the wait between inputs might be too long and the UI of the recogniser should be considered to be removed entirely. For the purposes of this example though I feel it nicely indicates when to speak, shows a moment of full screen render etc.

Conclusion

The ability to use native code inside Windows Phone 8 opens a new horizon for game developers. An effective code and potential middleware libraries usage can be used to enhance the uses experience to levels never experienced before. The Speech Recognition gives developers a whole new dimension to think about the way the user interacts with the app.

Links

source code File:VoiceMaze source.zip

 (C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.