DirectX on Windows Phone: 2D Game Example using DirectX Toolkit

This article describes how to start making 2D game applications in WP8 with native C++ and DirectX.

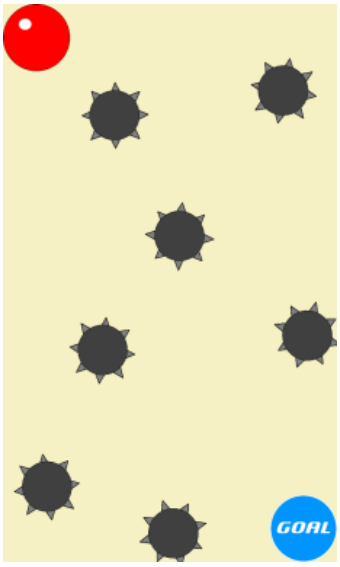
This article needs to be updated: If you found this article useful, please fix the problems below then delete the {{ArticleNeedsUpdate}} template from the article to remove this warning.

Reasons: hamishwillee
(17 Apr 2013)
This article was created using the RC version of the SDK and needs to be updated to the current SDK version. There are known link errors in compiling code at the end of part 3 (see comments for details).

Introduction

With Direct2D being removed from the Windows Phone 8 edition of DirectX, game developers will need another alternative for developing 2D games. Using XNA to make a 2D-game is one option, but it isn't necessarily the best one if the developer prefers C++ features instead of C# or if the developer already has a DirectX game made for desktop in C++ and wants to port that to WP8. Also, when using XNA you must target Windows Phone 7.1, and that will mean you can't use most of the Windows Phone 8 specific functionality. A viable option is the DirectX Toolkit (DirectXTK or DXTK) which makes 2D graphics rendering easier and still uses C++ and DirectX APIs to do it. We are going to be using DirectXTK for 2D graphics rendering and make a small example game that utilizes the WP8 devices' accelerometer sensor.

In the end of the example, the application is going to look like this:



DirectX Toolkit

DirectX Toolkit is a collection of helper classes that help the developer to write DirectX 11 code easier. In this article we are going to be using three of the toolkit's helper classes:

- **DDSTextureLoader:** Like the name suggests, this function loads DDS textures from either a file in a memory buffer or a .dds file from disk. It creates a Direct3D 11 resource and a Direct3D 11 shader resource view for the loaded texture and the developer can then use these resources how he chooses.
- **SpriteBatch:** Originally used in XNA Game Studio, the SpriteBatch has been ported to the Direct3D 11 C++ environment with very similar functionality. It includes a Draw()-method similar to the one found in OpenGL ES with various overloads. We will be using this Class to render the graphics in our game example.
- **SpriteFont:** Also ported from XNA to C++, the SpriteFont-class renders .spritefont bitmap files. It comes with a MakeSpriteFont command line tool for creating spritefonts.

DirectXTK also has several other features useful for DirectX game development which we aren't using in this example. For further reading you can check out the DirectXTK Project homepage.

2D Game Example Part 1: Creating a new VS Project, modifying the template and setting up DirectXTK

Modifying the Template

To start programming our game example, first we are going to create a new Visual Studio project from the WP8 Direct3D Application template. Some of the generated code and files aren't necessary in the example so first we are going to be removing some unused code from the files and also deleting and renaming others. Creating a project from the template and modifying it is still considerably faster and easier than creating everything from scratch.

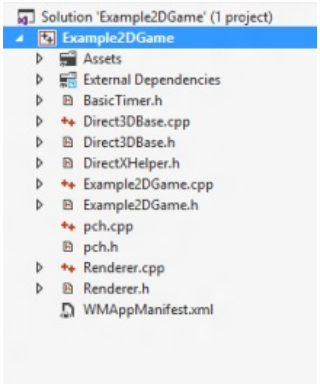
 (C) Copyright Nokia 2013. All rights reserved.

To begin, let's create a new VS project from the Direct3D Application template. The project creation process is explained in the Windows Phone Native C++ and DirectX - First Direct3D App, setting up Touch and Sensors article if you aren't familiar with it. Let's name the new project Example2DGame.

First up, the SpriteBatch renderer doesn't require any shaders, so we can safely delete the PixelShader.hlsl and VertexShader.hlsl files.

Next up, we are going to rename the CubeRenderer.h and CubeRenderer.cpp files. Instead of rendering a cube, these files will include the rendering for all the objects in the example game, so let's just rename the files Renderer.h and Renderer.cpp. To also change the class name, use the "Find and Replace"-tool to rename all CubeRenderer occurrences in the code to Renderer (remember to set the find scope to "Entire Solution").

Our project structure should now look like this:



Next, download this Textures zip-file and extract it's contents to the Assets folder in your Visual Studio project folder. After this, in Visual Studio, right-click your Example2DGame project and click Add -> Existing item, select the three files from the Assets-folder and click OK. The files should now be visible in the project tree under the project.

Next we are going to remove some not needed code from the Renderer.h and Renderer.cpp files

Things to remove from Renderer.h:

- ModelViewProjectionConstantBuffer and VertexPositionColor structs
- Private class members m_inputLayout, m_vertexShader, m_pixelShader, m_constantBuffer, m_vertexBuffer, m_indexCount, m_constantBufferData

Our Renderer.h should then look like this:

```
#pragma once

#include <wrl\client.h>
#include <memory>

#include "Direct3DBase.h"
#include <DirectXMath.h>

ref class Renderer sealed : public Direct3DBase
{
public:
    Renderer();

private:
    ~Renderer();

public:
    virtual void CreateDeviceResources() override;
    virtual void CreateWindowSizeDependentResources() override;
    virtual void Render() override;
    void Update(float timeTotal, float timeDelta);

private:
    bool m_loadingComplete;
    uint32 m_indexCount;
};
```

Next we remove some not needed code from Renderer.cpp:

In the CreateDeviceResources() method you can leave the Direct3DBase::CreateDeviceResources(); in the first line and remove everything else from inside the method. The method should then look like this:

```
void Renderer::CreateDeviceResources()
{
    Direct3DBase::CreateDeviceResources();

}
```

You can do the same in CreateWindowSizeDependentResources() method: only leave Direct3DBase::CreateWindowSizeDependentResources(); at the top and remove all other code from the method:

```
void Renderer::CreateWindowSizeDependentResources()
{
    Direct3DBase::CreateWindowSizeDependentResources();

}
```

In the Update() method, you can remove all its contents, leaving it empty for no, which means it plainly looks like this:

 (C) Copyright Nokia 2013. All rights reserved.

```
void Renderer::Update(float timeTotal, float timeDelta)
{

}
```

From the Render() method, remove all but the background clearing method call m_d3dContext->ClearRenderTargetView, the m_d3dContext->ClearStepthStencilView call from the beginning of the method and the m_d3dContext->OMSetRenderTargets() call from the end. We can also change the background clearing slightly so that the background has a bit brighter color (default is dark blue). Update() should then look like this:

```
void Renderer::Render()
{
    // Clear background with bg_color
    const float bg_color[] = { 245.0f/255.0f, 241.0/255.0f, 196.0f/255.0f, 1.000f };
    m_d3dContext->ClearRenderTargetView(
        m_renderTargetView.Get(),
        bg_color
        );

    m_d3dContext->ClearDepthStencilView(
        m_depthStencilView.Get(),
        D3D11_CLEAR_DEPTH,
        1.0f,
        0
        );

    m_d3dContext->OMSetRenderTargets(
        1,
        m_renderTargetView.GetAddressOf(),
        m_depthStencilView.Get()
        );
}
```

This is all the code removal we need to do. Next up we set up the DirectX Toolkit.

Setting up DirectX Toolkit

To install DXTK, first we need the Visual Studio project of the toolkit. In desktop applications, you could just download a newest version from the project homepage, but as of writing this article, the Phone version of the DXTK isn't yet available at the site. So we've uploaded it for you to download here.
Update: The DirectXTK Homepage now has an updated version of the toolkit with support to Windows Phone 8, so you can also download it from there. Make sure you use the "DirectXTK_WindowsPhone8"- Visual Studio project included in the package when adding the dependencies.

It takes a few steps to add the DXTK project to our game example solution:

1. In the VS solution view, right click on the solution and select Add -> Existing project

2. Navigate to the downloaded DirectXTK folder and select the DirectXTK_11_Phone-project

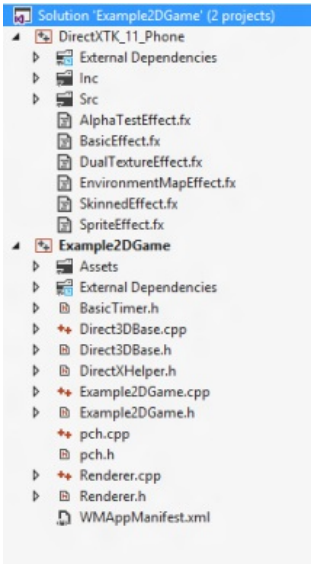
3. Right click on the Example2DGame project and select References... . In the references settings window, click “Add New Reference...” and select the DirectX Toolkit from the list and click OK. Click OK again to close the property page.

4. Right click on the Example2DGame project and select Project Dependencies. In the dependency window, make sure that the DirectX Toolkit is selected.

5. Right click on Example2DGame and select “Properties”. On the left side of the window navigate to C/C++ -> General. In the main window click on the Additional Include folders and select “Edit”. Click on the new folder button and a new line appears on the list. Use the Browse button (“...”) on the end of the line to navigate to the DirectXTK folder and from that folder select the “Inc” folder and click OK.

6. Finally, in the project Properties, go to Linker -> Input and select Edit.. on Additional Dependencies. Add dxguid.lib and XAudio2.lib to the list and click OK until the Properties window is close

You should now be able to include the DXTK header files in your code files and the project structure should now look like this:



Now that we’ve included the DXTK in our solution, we can finally start programming the actual game.

 (C) Copyright Nokia 2013. All rights reserved.

2D Game Example Part 2: Adding the Movable Ball, Getting Move Directions from the Device Accelerometer

In this part of the article, we are going to add our first rendered object - the ball the player moves around - to our game example. We are also going to set up the accelerometer sensor and make the ball move based on the device's movement.

Adding the Ball class

First, let's create a new class for the movable ball to keep the code more clear. Let's create a new header and cpp file for the class by right clicking the Example2DGame project and selecting Add -> new file. From the window, select Code -> Header file / Source file accordingly. Name the files Ball.h and Ball.cpp

So, let's start adding some code! Let's go to the Ball.h file first. For public functions, in addition to the default constructor, we are going to make a constructor that takes the x and y coordinates of the starting position of the ball as parameters, as well as the window dimensions of the device screen, which are then used to calculate the size of the ball. For now, we also make a Draw()-method, that takes a SpriteBatch class pointer as a parameter and a loadTexture() method which takes a pointer of an ID3D11Device class as a parameter. We will be adding a few class methods later on but for now we only need these.

We are also going to add a diameter float as a public member of the class, so we can use it further on in other parts of the program for various calculations as well as two floats for the x and y position of the ball.

As private class members we have ID3D11ShaderResourceView pointer for out texture and a scale float used to scale the ball rendering to different sized device screens.

Also include SpriteBatch.h and DDSTextureLoader.h from the DXTK and D3D11.h from the SDK library as well as namespaces DirectX, std and Windows::Foundation.

So, this is how our Ball.h looks at the moment:

```
#pragma once

#include "SpriteBatch.h"
#include "DDSTextureLoader.h"
#include <D3D11.h>

using namespace DirectX;
using namespace std;

class Ball
{
public:
 Ball();
 Ball(float x, float y, Rect* windowBounds);
 void Draw(SpriteBatch* sb);
 void loadTexture(ID3D11Device* d3dDevice);
 float posX;
 float posY;
 float diameter;

private:
 ID3D11ShaderResourceView* m_Texture;

 float scale;
};
```

Let's move to Ball.cpp and add the definitions.

The default constructor isn't going to be used in our code, so we will leave that blank. In the overloaded constructor that takes parameters, we are going to set the posX and posY members according to the given parameters and calculate the drawing scale and diameter according to the window bounds of the device. We also format the texture pointer as null.

In the loadTexture() method, we simply call the CreateDDSTextureFromFile method found in the DXTK and give it the d3dDevice, m_Texture and the texture file name as parameters (and also a nullpointer in place of a ID3DResource parameter which isn't needed in this example) the method then binds the texture to the m_Texture member for later use.

The Draw() method calls the Draw method of the spritebatch class it receives as a parameter. The method has various overloads with different rendering parameters and from those, we are going to use the following:

```
void Draw(_In_ ID3D11ShaderResourceView* texture, XMFLOAT2 const& position, _In_opt_ RECT const* sourceRectangle, FXMVECTOR color, f
```

Remember to add an include to Ball.h in the beginning of the file. Also, add an include to pch.h before that. Visual Studio requires that phc.h is included in the beginning of every file, otherwise build errors will appear. Also add a using to namespace Windows::Graphics::Display for getting the LogicalDpi of the screen.

So this is how our Ball.cpp will look like:

```
#include "pch.h"
#include "Ball.h"

using namespace Windows::Graphics::Display;

Ball::Ball()
{
}

Ball::Ball(float x, float y, Rect* windowBounds)
```

 (C) Copyright Nokia 2013. All rights reserved.

```
{
 posX = x;
 posY = y;

 scale = DisplayProperties::LogicalDpi / 96.0f;

 //Set the diameter of the ball to 1/5th of the screen width
 diameter = windowBounds->Width * scale / 5.0f;

 m_Texture = nullptr;
}

void Ball::loadTexture(ID3D11Device* d3dDevice)
{
 CreateDDSTextureFromFile(d3dDevice, L"ball.dds", nullptr, &m_Texture, MAXSIZE_T);
}

void Ball::Draw(SpriteBatch* sb)
{
 sb->Draw(m_Texture, XMFLOAT2(posX, posY), nullptr, Colors::White, 0.0f, XMFLOAT2(250.0f, 250.0f), XMFLOAT2(diameter / 500.0f, diame
}
```

Now we have the ball class ready! Next up we are going to add the code necessary to draw the ball and move it around with the accelerometer sensor.

Drawing the Ball on Screen and Moving it with Accelerometer Sensor

Before we start adding code, let's enable the device sensor permissions for the accelerometer from the WMAppManifest.xml file. Open the file, go to Capabilities tab and check the ID_CAP_SENSORS capability. After this, save the file.

To draw the ball on the screen, we first need to initialize a SpriteBatch, Accelerometer and AccelerometerReading class objects in our Renderer class. First, to the Renderer.h, we add the declarations. The DXTK documentation recommends using SpriteBatch with a unique_ptr, so we declare it with that in the private members. We also need to add an include for the SpriteBatch.h header file and a using for the DirectX, Windows::Devices::Sensors and std namespaces. We are also going to add a scale float and a Ball object pointer from the newly created Ball class (include Ball.h):

```
#pragma once

#include <wrl\client.h>
#include <memory>

#include "Direct3DBase.h"
#include <DirectXMath.h>

#include "SpriteBatch.h"

#include "Ball.h"

ref class Renderer sealed : public Direct3DBase
{
public:
    Renderer();

private:
    ~Renderer();

public:
    virtual void CreateDeviceResources() override;
    virtual void CreateWindowSizeDependentResources() override;
    virtual void Render() override;
    void Update(float timeTotal, float timeDelta);

private:
    bool m_loadingComplete;
    uint32 m_indexCount;

    unique_ptr<SpriteBatch> m_spriteBatch;

    Accelerometer^ m_accelerometer;
    AccelerometerReading m_accReading;

    float scale;

    Ball* m_ball;
};
```

Let’s move to the Renderer.cpp. Here, in the constructor we are going to format the m_accelerometer pointer to get the devices default accelerometer sensor with the static GetDefault() method of the Accelerometer class. We also calculate the scale with the devices display properties (need to add a using to namespace Windows::Graphics::Display).

```
Renderer::Renderer() :
    m_loadingComplete(false),
    m_indexCount(0)
{
    // Returns accelerometer ref if there is one; nullptr otherwise.
    m_accelerometer = Windows::Devices::Sensors::Accelerometer::GetDefault();

    scale = DisplayProperties::LogicalDpi / 96.0f;
}
```

In CreateDeviceResources() we are going to format the m_spriteBatch object with the m_d3dContext as a parameter:

```
void Renderer::CreateDeviceResources()
{
    Direct3DBase::CreateDeviceResources();

 m_spriteBatch = unique_ptr<SpriteBatch>(new DirectX::SpriteBatch(m_d3dContext.Get()));
}
```

In CreateWindowSizeDependentResources() we format the m_ball to point to a new Ball object with 0,0 as starting coordinates and pass the template-created m_windowBounds to it. After this we call the m_ball’s loadTexture():

 (C) Copyright Nokia 2013. All rights reserved.

```
void Renderer::CreateWindowSizeDependentResources()
{
    Direct3DBase::CreateWindowSizeDependentResources();

 ball = new Ball(0.0f, 0.0f, &m_windowBounds);
 ball->loadTexture(m_d3dDevice.Get());
}
```

Next, in the Update() method, we start getting the accelerometer readings by storing them to the m_accReading member and then move the ball according to the reading. We also have a bunch of else-if statements to make sure that the ball doesn't move over the screen bounds:

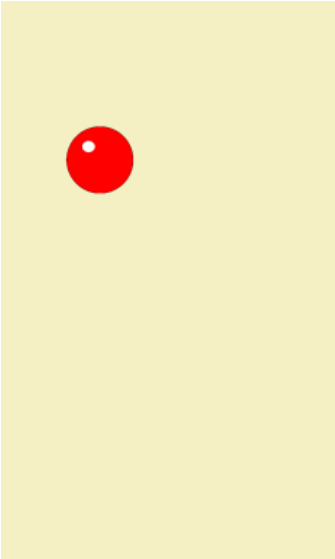
```
void Renderer::Update(float timeTotal, float timeDelta)
{
 if (m_accelerometer != nullptr)
 {
  m_accReading = m_accelerometer->GetCurrentReading();
 }

 // The 4.0f multiplier value is to add speed to the ball
 ball->posX += (float)m_accReading->AccelerationX * 4.0f * scale;
 // -1 multiplier to flip the Y-axis
 ball->posY += (float)m_accReading->AccelerationY * 4.0f * scale *(-1);


 // Following ifs are to ensure the ball doesn't go over screen bounds
 if(ball->posX > m_windowBounds.Width * scale - ball->diameter/2.0f)
 {
  ball->posX = m_windowBounds.Width * scale - ball->diameter/2.0f;
 }
 else if(ball->posX < ball->diameter/2.0f)
 {
  ball->posX = ball->diameter/2.0f;
 }
 if(ball->posY > m_windowBounds.Height * scale - ball->diameter/2.0f)
 {
  ball->posY = m_windowBounds.Height * scale - ball->diameter/2.0f;
 }
 else if(ball->posY < ball->diameter/2.0f)
 {
  ball->posY = ball->diameter/2.0f;
 }

}
```

Finally, to the Render() method, we add SpriteBatch method calls to render the ball. To signal the SpriteBatch to start rendering, we first have to call the class' Begin() method. After this we can call the Draw()-method of the ball object with the m_spriteBatch as the parameter. Finally we call the End() method of the SpriteBatch to signal that the rendering for this pass is over:

```
void Renderer::Render()
{
    // Clear background with bg_color
    const float bg_color[] = { 245.0f/255.0f, 241.0/255.0f, 196.0f/255.0f, 1.000f };
    m_d3dContext->ClearRenderTargetView(
        m_renderTargetView.Get(),
        bg_color
        );

    m_d3dContext->ClearDepthStencilView(
        m_depthStencilView.Get(),
        D3D11_CLEAR_DEPTH,
        1.0f,
        0
        );

 m_spriteBatch->Begin();

 ball->Draw(m_spriteBatch.get());

 m_spriteBatch->End();

    m_d3dContext->OMSetRenderTargets(
        1,
        m_renderTargetView.GetAddressOf(),
        m_depthStencilView.Get()
        );
}
```

All the necessary code should now be in place for the ball to be drawn on the screen and be moved with the accelerometer. Go ahead and try to run it on your emulator or device. It should look like this:

 (C) Copyright Nokia 2013. All rights reserved.



In the next part we will be adding enemy objects to the game area.

2D Game Example Part 3: Adding Enemies

Next up, we add some enemies to the game area which the player ball has to avoid from touching.

The Enemy Class

First we create an Enemy class similar to the Ball class with its own Draw() and loadTexture() methods. Differing from the Ball class, we are going to make the ID3D11ShaderResourceView texture member and the loadTexture() method static because we will be creating multiple enemy objects and all of them have the same texture. We are also going to have a rotation float to make a simple rotation animation for the enemy.

So here's what the Enemy.h looks like:

```
#pragma once

#include "SpriteBatch.h"
#include "DDSTextureLoader.h"
#include <D3D11.h>

using namespace DirectX;
using namespace Windows::Foundation;

class Enemy
{
public:
 Enemy();
 Enemy(float x, float y, Rect* windowBounds);
 static void loadTexture(ID3D11Device* d3dDevice);
 void Draw(SpriteBatch* sb);

 static ID3D11ShaderResourceView* m_Texture;
 float posX;
 float posY;
 float diameter;

private:
 float rotation;
 float scale;

};
```

The Enemy.cpp is very similar to Ball.cpp. We set up the position, calculate the scale and the diameter of the enemy object and also randomize the rotation position in the overloaded constructor. The Draw() method is similar to the one in the Ball class, but in addition, after every drawing sequence, we increase the rotation value to keep the object rotating. The contents of the loadTexture() method are exactly the same as in Ball.cpp, we only give a different DDS texture filename as a parameter.

Enemy.cpp:

```
#include "pch.h"
#include "Enemy.h"

// Initialize static texture member
ID3D11ShaderResourceView* Enemy::m_Texture = nullptr;

using namespace Windows::Graphics::Display;

Enemy::Enemy()
{
}

Enemy::Enemy(float x, float y, Rect* windowBounds)
{
 posX = x;
 posY = y;

 scale = DisplayProperties::LogicalDpi / 96.0f;

 // Enemy diameter = 1/5th of screen width
```

 (C) Copyright Nokia 2013. All rights reserved.

```
 diameter = windowBounds->Width * scale / 5.0f;

 // Randomize starting rotation angle
 rotation = (float)rand() / ((float)RAND_MAX / 360.0f);
}

void Enemy::Draw(SpriteBatch* sb)
{
 sb->Draw(Enemy::m_Texture, XMFLOAT2(posX, posY), nullptr, Colors::White, XMConvertToRadians(rotation), XMFLOAT2(250.0f, 250.0f), XM

 rotation += 1.0f;
 if(rotation > 360.0f)
  rotation = 0.0f;
}

void Enemy::loadTexture(ID3D11Device* d3dDevice)
{
 CreateDDSTextureFromFile(d3dDevice, L"enemy.dds", nullptr, &Enemy::m_Texture, MAXSIZE_T);
}
```

Now we have the Enemy class ready. Next we implement the code that adds them to the playing area.

Adding Enemies to the Game Area

Next up we add the code to the Renderer class to draw the enemy objects on the screen.

First let's go to Renderer.h and add some declarations. In public methods, let's add two:

```
void randomizeEnemies();
void resetGame();
```

In randomizeEnemies() we add enemies to random locations of the game area and in resetGame(), we reset the game if the player ball collides with an enemy.

In private members we add a list containing Enemy class pointers and an iterator for the list. We also add an enemyCount integer to set how many enemies there are:

```
int enemyCount;
list<Enemy*> enemies;
list<Enemy*>::iterator iterator_enemies;
```

Also, remember to add includes for the Enemy.h header file and for the standard list library. This is how the Renderer.h file should now look like in its entirety:

```
#pragma once

#include <wrl\client.h>
#include <memory>

#include "Direct3DBase.h"
#include <DirectXMath.h>

#include "SpriteBatch.h"

#include "Ball.h"
#include "Enemy.h"

#include <list>

using namespace DirectX;
using namespace Windows::Devices::Sensors;
using namespace std;

ref class Renderer sealed : public Direct3DBase
{
public:
    Renderer();

private:
    ~Renderer();

public:
    virtual void CreateDeviceResources() override;
    virtual void CreateWindowSizeDependentResources() override;
    virtual void Render() override;
    void Update(float timeTotal, float timeDelta);
 void randomizeEnemies();
 void resetGame();

private:
    bool m_loadingComplete;

    uint32 m_indexCount;

 unique_ptr<SpriteBatch> m_spriteBatch;

 Accelerometer^ m_accelerometer;
 AccelerometerReading^ m_accReading;

 float scale;

 Ball* ball;

 int enemyCount;
 list<Enemy*> enemies;
 list<Enemy*>::iterator iterator_enemies;
};
```

Now we can move to Renderer.cpp and add some code there. First up, in the constructor, we only set the enemyCount to 7 (this is the optimal amount of

 (C) Copyright Nokia 2013. All rights reserved.

enemies that fit on the screen). After this we load the static texture for the Enemy objects by adding the following line to the CreateDeviceResources() method:

```
Enemy::loadTexture(m_d3dDevice.Get());
```

Next we can define the randomizeEnemies() method. Here we are going to do a for-loop that adds seven Enemy objects to the enemies list. Before every adding, we randomize the x and y coordinates of the enemy and check that those coordinates don't make the enemy being added overlap any previous enemy locations or the player location:

```
void Renderer::randomizeEnemies()
{
 float distX = 0.0f;
 float distY = 0.0f;

 bool positionsOverlap = false;

 for(int i = 0; i < 7; i++)
 {
  float x = 50.0f + (float)rand() / ((float)RAND_MAX / (m_windowBounds.Width * scale - 50.0f));
  float y = 50.0f + (float)rand() / ((float)RAND_MAX / (m_windowBounds.Height * scale - 50.0f));

  //Check that we don't create a new enemy too near to an excisting one
  for(iterator_enemies = enemies.begin(); iterator_enemies != enemies.end(); iterator_enemies++)
  {
   distX = x - (*iterator_enemies)->posX;
   distY = y - (*iterator_enemies)->posY;

   // Use ball diameter as helper value since the enemies' diameter is going to be the same
   if(((distX * distX) + (distY * distY)) <= (ball->diameter * 2) * (ball->diameter * 2))
   {
    positionsOverlap = true;
   }
  }

  // If random coordinates overlap the ball coordinates, we randomize them again, otherwise we create a new enemy at the coordinates
  if(!(x < ball->posX + ball->diameter && x > ball->posX - ball->diameter &&
   y < ball->posY + ball->diameter && y > ball->posY - ball->diameter) &&
   !positionsOverlap)
   enemies.push_back(new Enemy(x, y, &m_windowBounds));
  else
  {
   i--;
   positionsOverlap = false;
  }
 }
}
```

To make the rand() method work, we need to add a #include <time.h> at the beginning of the file and a srand((unsigned)time(0)); in the constructor.

Now let's add a call to randomizeEnemies() to the CreateDeviceResources() method:

```
void CreateDeviceResources()
{
    ...

    randomizeEnemies();
}

In the definition of the resetGame() function, we just clear the enemy list, randomize enemies again and move the player ball back t
<code cpp>
void Renderer::resetGame()
{
    enemies.clear();
    randomizeEnemies();
    ball->posX = 50.0f;
    ball->posY = 50.0f;
}
```

What we have left to do now, is to add collision detection to the Ball class and draw the enemies on the screen.

Adding Collision Detection and Enemy Drawing

First up is the collision detection. Basically, we are going to add a Boolean returning method to the Ball class that takes the enemy list as a parameter and checks for any overlapping coordinates, returning true if there are any.

Here's the declaration of the method which we will add to Ball.h:

```
bool collisionWithEnemy(list<Enemy*>* enemies);
```

We also need to add a list iterator that goes through the enemy list as a private member:

```
list<Enemy*>::iterator iterator_enemies;
```

No further declarations are needed at the moment, so we move to Ball.cpp for the definition of the collisionWithEnemy() method. In the method, we compare the positions of the ball and each enemy, taking into consideration their dimensions. We return a true if there is an overlap:

```
bool Ball::collisionWithEnemy(list<Enemy*>* enemies)
{
 for(iterator_enemies = enemies->begin(); iterator_enemies != enemies->end(); iterator_enemies++)
 {
  float distX = posX - (*iterator_enemies)->posX;
  float distY = posY - (*iterator_enemies)->posY;

  if((distX * distX) + (distY * distY) <= ((*iterator_enemies)->diameter) * ((*iterator_enemies)->diameter))
```

 (C) Copyright Nokia 2013. All rights reserved.

```
    return true;
  }

  return false;
}
```

Next up, we move to Renderer.cpp and add the collision detection to the Update() function. In the bottom of the function we add the following line:

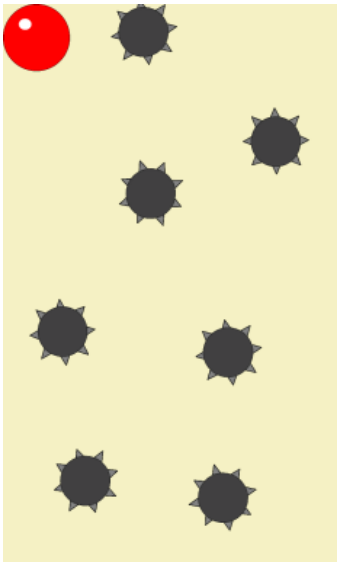
```
if(ball->collisionWithEnemy(&enemies))
{
    resetGame();
}
```

Now all we have left to do is the drawing of the enemies. For this we simply add a for-loop to the Render() method between the m_spriteBatch->Begin() and m_spriteBatch->End() lines where we go through the enemies list and call the Draw() method of each of the list entry:

```
for(iterator_enemies = enemies.begin(); iterator_enemies != enemies.end(); iterator_enemies++)
{
    (*iterator_enemies)->Draw(m_spriteBatch.get());
}
```

You can now try running the game on your emulator or device. All the seven enemy objects should appear inside the screen without any overlaps with themselves or the player ball. If you start moving the ball and try to touch an enemy, the game should reset the ball to the top left corner of the screen and randomize new enemy positions.

This is the current state of our game example:



That's the end of this part. In the final part of this article, we are going to render some text on the screen using the SpriteFont from the DXTK and add a goal object.

2D Game Example Part 4: Adding a Goal Object, Rendering Text

In the final part of this article, we are going to be adding a stationary goal object to the playing area of the game and also take a look at how to render text on the screen by implementing a simple "Tap to start"-functionality.

Adding the Goal

The goal object is going to be very similar to the enemy and ball objects. Unlike them, it is also going to be stationary in the lower right corner of the screen.

Like with the Enemy and Ball classes we first add Goal.h and Goal.cpp files to our project.

The Goal.h file will have same contents as the Enemy.h has. The constructor also only takes the window bounds as parameters as we automatically set the position to be in the lower right corner of the screen. We also have the usual Draw() and loadTexture() methods, ID3D11ShaderResourceView pointer for the texture, public diameter and position floats, and a private scale float.

Here's the Goal.h:

```
#pragma once

#include "SpriteBatch.h"
#include "DDSTextureLoader.h"
#include <D3D11.h>

using namespace DirectX;
using namespace std;
using namespace Windows::Foundation;
```

 (C) Copyright Nokia 2013. All rights reserved.

```
class Goal
{
public:
 Goal();
 Goal(Rect* windowBounds);
 void Draw(SpriteBatch* sb);
 void loadTexture(ID3D11Device* d3dDevice);
 float posX;
 float posY;
 float diameter;

private:
 ID3D11ShaderResourceView* m_Texture;

 float scale;

};
```

In the Goal.cpp we do the same stuff as in Enemy.cpp exclusind the rotation. In the constructor, we set the positions and calculate first the screen scale and then the diameter of the object according to it. We also automatically calculate the position to be in the lower right corner. The Draw() and loadTexture() methods are identical to the Enemy and Ball ones, only a different filename for the source texture.

Goal.cpp:

```
#include "pch.h"
#include "Goal.h"

using namespace Windows::Graphics::Display;

Goal::Goal()
{

}

Goal::Goal(Rect* windowBounds)
{
 scale = DisplayProperties::LogicalDpi / 96.0f;

 diameter = windowBounds->Width * scale / 5.0f;

 // Puts the goal in the lower left corner of the screen
 posX = windowBounds->Width * scale - diameter / 2.0f;
 posY = windowBounds->Height * scale - diameter / 2.0f;
}

void Goal::Draw(SpriteBatch* sb)
{
 sb->Draw(m_Texture, XMFLOAT2(posX, posY), nullptr, Colors::White, XMConvertToRadians(rotation), XMFLOAT2(250.0f, 250.0f), XMFLOAT2(
}

void Goal::loadTexture(ID3D11Device* d3dDevice)
{
 CreateDDSTextureFromFile(d3dDevice, L"goal.dds", nullptr, &m_Texture, MAXSIZE_T);
}
```

Now we have to add another collision detection method in the Ball class to check if it touches the Goal. In the Ball.h, we add the following public method declaration:

```
public:
    ...
    bool collisionWithGoal(Goal* goal);
```

Also, remember to add an include to Goal.h.

Now we define it in the Ball.cpp. We use the same kind of code as in the enemy collision detection. Only this time we won't go through a list as there is only one goal object:

```
bool Ball::collisionWithGoal(Goal* goal)
{
 float distX = posX - goal->posX;
 float distY = posY - goal->posY;

 if(distX * distX + distY * distY <= goal->diameter * goal->diameter)
  return true;

 return false;
}
```

Next up, we add code to Renderer.h and Renderer.cpp. In the header file we simply, add a Goal pointer as a private member (remember to add an include to Goal.h):

```
private:
    ...
    Goal* goal;
```

In Renderer.cpp's CreateWindowSizeDependentResources() we add two lines before the randomizeEnemies() method call to initialize the goal pointer and to load it's texture:

```
void Renderer::CreateWindowSizeDependentResources()
{
    Direct3DBase::CreateWindowSizeDependentResources();

 ball = new Ball(0.0f, 0.0f, &m_windowBounds);
 ball->loadTexture(m_d3dDevice.Get());

 goal = new Goal(&m_windowBounds);
 goal->loadTexture(m_d3dDevice.Get());

 randomizeEnemies();
```

 (C) Copyright Nokia 2013. All rights reserved.

```
}
```

To the Update() function we add the code to detect ball collision with the goal. For this example, like we do with the enemy collision, we simply just reset the game when the collision with the goal happens. If this were a proper game, we could for example execute code that loads a new level with different amount of enemies when the collision happens.

```
void Renderer::Update(float timeTotal, float timeDelta)
{
    ...

    if(ball->collisionWithGoal(&goal))
    {
        OutputDebugStringA("goal collision!\n");
        resetGame();
    }
}
```

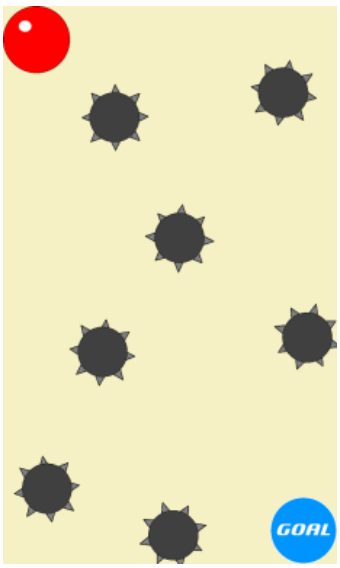
We also need to add code to the randomizeEnemies() method so that the enemies won't appear on top of the goal. This means adding some conditions to the last if-sentence of the method so it will look like this:

```
void Renderer::randomizeEnemies()
{
 ...

  // If random coordinates overlap the goal or ball coordinates, we randomize them again, otherwise we create a new enemy at the coo
  if(!(x < goal->posX + goal->diameter && x > goal->posX - goal->diameter &&
   y < goal->posY + goal->diameter && y > goal->posY - goal->diameter) &&
   !(x < ball->posX + ball->diameter && x > ball->posX - ball->diameter &&
   y < ball->posY + ball->diameter && y > ball->posY - ball->diameter)&& !positionsOverlap)
   enemies.push_back(new Enemy(x, y, &m_windowBounds));
  else
  {
   i--;
   positionsOverlap = false;
  }
 }
}
```

Lastly we add a line to the Render() method where we draw the goal to the screen. As with the Ball and Enemy objects, the line goes between m_spriteBatch->Begin() and End():

```
void Renderer::Render()
{
    ...
    goal->Draw(m_spriteBatch.get());
}
```

Now you can try running the app on your device or emulator. There should now be a goal object in the lower right corner of the screen and when the ball collides with it, the game should reset.



Rendering Text

To finish up this example, we are going to do some simple text rendering. We will add a “Tap to start!” text in the middle of the screen when the game is started and when a game reset happens. The player will start the game by tapping on the screen.

For text rendering we will be using the SpriteFont class from the DXTK. The basic functionality of the class is to load a .spritefont file that includes all the needed font info and then call the class’s DrawString() function which works similarly to the SpriteBatch’s Draw-method.

Creating a .spritefont involves a few steps.

First we open the MakeSpriteFont Visual Studio project in the DXTK folder and build it. Now there should be an executable file in the output directory. We are going to execute it in the command prompt and give it parameters according to what kind of font we want. For this example, we make a normal Arial

 (C) Copyright Nokia 2013. All rights reserved.

using spritefont sized 50 pixels and save it to a file names myfont.spritefont with this kind of command:

MakeSpriteFont.exe "Arial" myfont.spritefont /FontSize:50

More about the command line options can be read from here.

Now we add that file to out Example2DGame project by right clicking the project and choosing Add->Existing file. When the file is added, we also need to mark it as project content by right clicking it, choosing properties and typing “yes” to the Content property.

First thing we add to the code is the declaration of the SpriteFont member in Renderer.h. Rememeber to also add an #include “SpriteFont.h” to the top of the file. Like with SpriteBatch, it’s recommended to use unique_ptr with SpriteFont:

```
private:
    ...
    unique_ptr<SpriteFont> m_spriteFont;
```

In the Renderer.cpp we initialize the m_spriteFont in the CreateDeviceResources() method:

```
void Renderer::CreateDeviceResources()
{
    ...
    m_spriteFont = unique_ptr<SpriteFont>(new SpriteFont(m_d3dDevice.Get(), L"myfont.spritefont"));
}
```

Now we just add the rendering code to the Render() method. The SpriteFont class has a handy MeasureString() method which measures the length of a given sentence with the specified font. We can use this method to center the Text in the screen. So in the Render() method, before we call the DrawString-method, we store the measured length of the text to a float pointer. Also, remember to put the added code between m_spriteBatch->Begin() and m_spriteBatch->End(). The DrawString() method overload we are using takes a bunch of parameters: first we need to pass it the pointer of the m_spriteBatch we are using, after that the wanted text and then like the SpriteBatch's Draw() method, we are going to pass different rendering parameters like the position, color and rotation of the text.

```
void Renderer::Render()
{
    m_spriteBatch->End();
    ...

    float* stringlength = m_spriteFont->MeasureString(L"Tap to start!").n.128_f32;
    m_spriteFont->DrawString(m_spriteBatch.get(), L"Tap to start!", XMFLOAT2(m_windowBounds.Width * scale / 2.0f, m_windowBounds.Hei

    m_spriteBatch->End();
}
```

Now if you run the app, you can see the "Tap to start!" text in the middle of the screen. In this point though, the text is visible all the time and the game runs straight from the start. So as a last thing, we are going to add a few lines of code to make the example work properly.

First we add one private Boolean member called gameStarted and public get and set methods to the Renderer.h file which return and set the value of this Boolean.

```
ref class Renderer sealed : public Direct3DBase
{
public:
    Renderer();

private:
    ~Renderer();

public:
    ...

    void setGameRunning(bool running);
    bool getGameRunning();
    void resetGame();

private:
    ...

    bool gameStarted;
};
```

Next we add code into the touch event handlers that came with the template in the Example2DGame.cpp to set the gameStarted Boolean to true when the player taps the screen. We put the code in the OnPointerReleased() method so it looks like this:

```
void _2DExample::OnPointerReleased(CoreWindow^ sender, PointerEventArgs^ args)
{
 if(m_renderer->getGameRunning() == false)
 {
  m_renderer->setGameRunning(true);
  m_renderer->randomizeEnemies();
 }

 OutputDebugString(L"tap");
}
```

We simply check if the game is running when a touch release happens. If it isn’t, we set the gameStarted Boolean to true and randomize enemies on the screen.

Now in the Renderer.cpp we just set the gameStarted to false in the constructor and define the two methods:

```
Renderer::Renderer() :
    m_loadingComplete(false),
    m_indexCount(0)
```

 (C) Copyright Nokia 2013. All rights reserved.

```
{
    ...
    gameStarted = false;
}

void Renderer::setGameRunning(bool running)
{
 gameStarted = running;
}

bool Renderer::getGameRunning()
{
 return gameStarted;
}
```

We also need to add some if-statements to the Render() function so that the enemies are drawn only when the gameStarted Boolean is true and the text is drawn only when it’s false. So, we add the following if-statements around the enemy drawing for-loop and around the text drawing:

```
void Renderer::Render()
{
        ...

 m_spriteBatch->Begin();

        // Draw enemies if game is running
 if(gameStarted)
 {
  for(iterator_enemies = enemies.begin(); iterator_enemies != enemies.end(); iterator_enemies++)
  {
   (*iterator_enemies)->Draw(m_spriteBatch.get());
  }
 }

 ball->Draw(m_spriteBatch.get());
 goal->Draw(m_spriteBatch.get());


 float* stringlength = m_spriteFont->MeasureString(L"Tap to start!").n128_f32;

        // Draw tap text if game is not running
 if(!gameStarted)
  m_spriteFont->DrawString(m_spriteBatch.get(), L"Tap to start!", XMFLOAT2(m_windowBounds.Width * scale / 2.0f, m_windowBounds.Heigh

 m_spriteBatch->End();

        ...
}
```

In addition, in the Update() method, we add an if-statement around the ball moving, so that the ball won’t move if the game isn’t started:

```
void Renderer::Update(float timeTotal, float timeDelta)
{
 ...

 if(gameStarted)
 {
  // The 4.0f multiplier value is to add speed to the ball
  ball->posX += (float)m_accReading->AccelerationX * 4.0f * scale;
  // -1 multiplier to flip the Y-axis
  ball->posY += (float)(m_accReading->AccelerationY) * 4.0f * scale *(-1);
 }

        ...
}
```

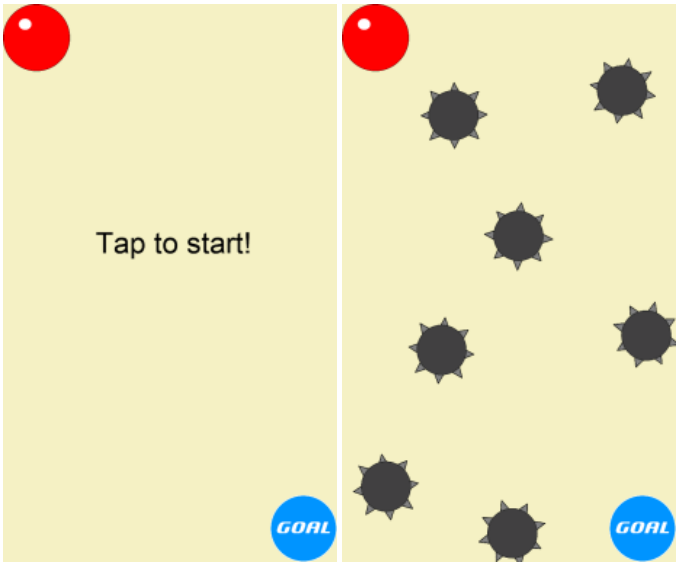
Now a small deletion from CreateDeviceResources(). From there we can remove the randomizeEnemies() method call since it's going to be done in the touch event handler when the player starts the game.

Lastly, a line needs to be added to the resetGame() method in Renderer.cpp where gameStarted is set to false when the game is reset. We can also remove the randomizeEnemies() method call since it's called in the touch event:

```
void Renderer::resetGame()
{
 enemies.clear();
 ball->posX = 50.0f;
 ball->posY = 50.0f;
 gameStarted = false;
}
```

That is for all the code! Our example is now ready. You can run it to see the final version. When it starts, the Ball and the Goal should be visible on the screen, along with the "Tap to start!" text. When you tap on the screen, the Ball becomes movable and the enemies appear on the screen. When you run into an enemy or the goal, the game resets, going back to the tap screen.

 (C) Copyright Nokia 2013. All rights reserved.



That is the end of this article. You should now have fairly good information on how to start developing a 2D game with native C++ and DirectX on Windows Phone.

Thank you for reading!

 (C) Copyright Nokia 2013. All rights reserved.

 (C) Copyright Nokia 2013. All rights reserved.