

الدرس الحادي عشر

أهلاً بكم في الدرس الحادي عشر من سلسلة دروس تعلم 3D Xna (السلسلة الثانية)، سنقوم في هذا الدرس بإضافة أشباح نقطية "Point Sprites"، و سوف نستخدم تقنية الـ "Billboarding" لتمثيل الرصاص في اللعبة.

لتمثيل الرصاص، بإمكاننا أن نستخدم كره حقيقية ثلاثية الأبعاد، ولكن ذلك سوف يؤدي إلى زيادة الجهد على بطاقة الرسومات. إذا بدلاً من ذلك، سوف نقوم باستخدام صورة كره نارية ثنائية الأبعاد بسيطة جداً، بإمكانك تنزيلها من [هذا الرابط](#) (أو من الملفات المرفقة في الدرس الأول). سوف يكون مطلوب منك فقط تحديد النقطة المركزية للصورة في الفضاء الثلاثي الأبعاد، و من ثم سوف تقوم الـ Xna برسم الصورة، بحيث تكون دائماً مواجهة للمشاهد بالإضافة إلى تحجيمها بحيث تعكس المسافة بين المشاهد و النقطة في الفضاء الثلاثي الأبعاد. تسمى هذه التقنية بالـ "Billboarding".

الصورة الثنائية الأبعاد أيضاً تسمى شبح "Sprite"، و بما أن الـ Xna يلزمها فقط إحداثيات نقطة المركز في الصورة كموقع ثلاثي الأبعاد، فإن هذه الأشباح ثنائية الأبعاد التي تستخدم في العالم الثلاثي الأبعاد تسمى بالأشباح النقطية "Point Sprites". بما أننا بحاجة إلى نقطة واحدة فقط، سنقوم بهذه الطريقة بتوفير الكثير من نقل البيانات عبر مدخل الـ PCI الخاص ببطاقة الرسومات.

عندما نبدأ بإطلاق النار، نريد أن تتحرك الرصاصات إلى الأمام بشكل متواصل. لذا، لكل رصاصة يجب أن نتابع كل من الموقع و الدوران الحالي من أجل حساب اتجاه الرصاصة، تماماً مثل الطائرة. لذلك سوف نقوم بتعريف تركيب "Struct" جديد، قم بكتابته في أعلى الكود لديك، فوق المتغيرات الخاصة بنا:

```
struct Bullet
{
    public Vector3 position;
    public Quaternion rotation;
}
```

بعد تعريف هذا التركيب، بإمكاننا أن ننشئ كائنات من النوع "Bullet". لكل كائن من النوع "Bullet"، بإمكاننا تخزين متجه ثلاثي الأبعاد "Vector3" و رباعي "Quaternion" من أجل تخزين الموقع و الدوران، على التوالي.

سوف نقوم بمتابعة قائمة تحتوي على كائنات الرصاصات تلك. أيضاً، لتحديد سرعة الإطلاق، نحن بحاجة لمعرفة الوقت الذي أطلقت فيه آخر رصاصة. أخيراً، سوف نحتاج إلى متغير من النوع Texture2D من أجل تخزين صورة الرصاصة. لذا قم بإضافة الأسطر التالية إلى تعريف المتغيرات لدينا:

```
Texture2D bulletTexture;

List<Bullet> bulletList = new List<Bullet> ();
double lastBulletTime = 0;
```

بإمكانك تنزيل ملف صورة الرصاصة من [هنا](#).

بطبيعة الحال قم بإستيراد الصورة إلى المشروع كما فعلنا سابقا. بعدها، سوف نقوم بتحميل هذه الصورة ثنائية الأبعاد إلى متغير الخامة. يتم ذلك من خلال إضافة السطر التالي إلى الدالة LoadContent:

```
bulletTexture = Content.Load<Texture2D> ("bullet");
```

الآن، في كل مره يقوم فيها المستخدم بالضغط على زر المسافة "Spacebar"، نريد أن يتم إنشاء رصاصة جديدة و إضافتها إلى قائمة الرصاص bulletList، لذا قم بإضافة الكود التالي في نهاية الدالة ProcessKeyboard():

```
if (keys.IsKeyDown(Keys.Space))
{
    double currentTime = gameTime.TotalGameTime.TotalMilliseconds;
    if (currentTime - lastBulletTime > 100)
    {
        Bullet newBullet = new Bullet();
        newBullet.position = xwingPosition;
        newBullet.rotation = xwingRotation;
        bulletList.Add(newBullet);

        lastBulletTime = currentTime;
    }
}
```

عندما يتم الضغط على زر المسافة، نقوم بمقارنة الوقت الحالي بالوقت الذي تم فيه إطلاق آخر رصاصة. إذا كانت الطلقة الأخيرة منذ أكثر من 100 ملي ثانية، سوف نقوم بإطلاق رصاصة جديدة.

إذا تحقق شرط إطلاق رصاصة جديدة، نقوم بإنشاء كائن جديد من النوع Bullet، و يتم إعطائها قيم الموقع و الدوران الحالي الخاصين بالطائرة. السبب في ذلك اننا نريد أن تطير الرصاصة بنفس الإتجاه الخاص بالطائرة ومن نفس مكان الطائرة في لحظة الإطلاق. السطر قبل الأخير يقوم بإضافة الرصاصة الجديدة إلى قائمة الرصاص bulletList.

قبل أن ننتقل إلى كود الرسم، دعنا ننتهي من موضوع تحريك الرصاص إلى الأمام. قمنا سابقا بإنشاء دالة تقوم بكل الحسابات: MoveForward. سنقوم بإنشاء دالة جديدة لإستدعائها، UpdateBulletPositions، حيث ستقوم بالمرور على قائمة الرصاص bulletList ومن ثم تحديث موقع كل رصاصة. الكود التالي:

```
private void UpdateBulletPositions(float moveSpeed)
{
    for (int i = 0; i < bulletList.Count; i++)
    {
        Bullet currentBullet = bulletList[i];
        MoveForward(ref currentBullet.position, currentBullet.rotation,
moveSpeed * 2.0f);
        bulletList[i] = currentBullet;
    }
}
```

جميل جدا. موقع كل رصاصة في قائمة الرصاص تم تحديثه. هذه الدالة سوف تستقبل أيضا نفس القيمة moveSpeed مثل الطائرة، لكن بما أننا نقوم بضرب القيمة بـ 2.0 فهذا يعني أن الرصاص سوف يطير بضعف سرعة الطائرة.

قم بإستدعاء هذه الدالة من داخل الدالة Update():

```
UpdateBulletPositions (moveSpeed) ;
```

بعدها، نريد أن نرسم الأشباح "Sprites" التي تمثل الرصاصات. حيث لكل رصاصة يلزم الـ Xna شيئين إثنين: موقع و حجم الشبح الذي يمثل تلك الرصاصة. سنستعمل تقنية جديدة من التقنيات الموجودة في ملف التأثير، تسمى "PointSprites"، حيث سوف نقوم بشكل أوتوماتيكي بتصغير حجم الصورة بناء على المسافة بينها وبين الكاميرا.

كما في السلسلة الأولى، نحن الآن في حالة حيث لا نستطيع إستخدام أحد تنسيقات الرؤوس "Vertex Format" المقدمة من الـ Xna. لذا سوف نقوم بتعريف تنسيق للرؤوس خاص بنا، وهو عبارة عن تركيب "Struct" قادر على تخزين البيانات اللازمة. سوف يتم مناقشة هذا الموضوع بالتفصيل في السلسلة الثالثة -إن شاء الله-. إذن ضع تعريف التركيب التالي في أعلى الكود، بعد تعريف تركيب الرصاصة "Bullet" مثلا:

```
private struct VertexPointSprite
{
    private Vector3 position;
    private float pointSize;

    public VertexPointSprite(Vector3 position, float pointSize)
    {
        this.position = position;
        this.pointSize = pointSize;
    }

    public static readonly VertexElement[] VertexElements =
    {
        new VertexElement(0, 0, VertexElementFormat.Vector3,
        VertexElementMethod.Default, VertexElementUsage.Position, 0),
        new VertexElement(0, sizeof(float)*3,
        VertexElementFormat.Single, VertexElementMethod.Default,
        VertexElementUsage.PointSize, 0),
    };
    public static int SizeInBytes = sizeof(float) * (3 + 1);
}
```

هذا التركيب بإمكانه تخزين البيانات التي نحتاجها الـ Xna من أجل رسم الأشباح النقطية "Point Sprites": الموقع و الحجم. قمنا أيضا بتعريف مصفوفة من عناصر الرؤوس "VertexElements"، حيث سوف نحتاجها من أجل إنشاء تعريف للرؤوس "VertexDefinition" مناسب لهذا النوع من الرؤوس. تذكر أننا نحتاج تعريف الرؤوس في مرحلة الرسم، حيث يوضح تعريف الرؤوس لبطاقة الرسومات أنواع البيانات التي تحتويها هذه الرؤوس بالضبط.

قم بإضافة الـ "VertexDeclaration" إلى المتغيرات في أعلى الكود:

```
VertexDeclaration pointSpriteVertexDeclaration;
```

ثم قم بمليء هذا المتغير من خلال إضافة السطر التالي في الدالة LoadContent(). وهو مكان مناسب حيث أننا بحاجة لتعريف الـ "VertexDeclaration" مره واحدة فقط، بما أن نوع البيانات المخزن في الرأس لن يتغير.

```
pointSpriteVertexDeclaration = new VertexDeclaration(device,
VertexPointSprite.VertexElements);
```

الآن سوف نقوم بتعريف دالة جديدة، بإسم DrawBullets، حيث سوف تقوم برسم الرصاصات المخزنة في قائمة الرصاص bulletList. تبدو صعبة بعض الشيء، ولكنها لا تحتوي على شيء لم نره من قبل:

```
private void DrawBullets()
{
    if (bulletList.Count > 0)
    {
        VertexPointSprite[] spriteArray = new
VertexPointSprite[bulletList.Count];
        for (int i = 0; i < bulletList.Count; i++)
            spriteArray[i] = new
VertexPointSprite(bulletList[i].position, 50);

        effect.CurrentTechnique = effect.Techniques["PointSprites"];
        Matrix worldMatrix = Matrix.Identity;
        effect.Parameters["xWorld"].SetValue(worldMatrix);
        effect.Parameters["xView"].SetValue(viewMatrix);
        effect.Parameters["xProjection"].SetValue(projectionMatrix);
        effect.Parameters["xTexture"].SetValue(bulletTexture);

        device.RenderState.PointSpriteEnable = true;

        effect.Begin();
        foreach (EffectPass pass in effect.CurrentTechnique.Passes)
        {
            pass.Begin();
            device.VertexDeclaration = pointSpriteVertexDeclaration;
            device.DrawUserPrimitives(PrimitiveType.PointList,
spriteArray, 0, spriteArray.Length);
            pass.End();
        }
        effect.End();

        device.RenderState.PointSpriteEnable = false;
    }
}
```

أولاً قمنا بفحص إذا كانت قائمة الرصاص غير فارغة. إذا كان هنالك بعض الرصاص لرسمه، تنتقل للتالي. من أجل رسم الأشكال، نحن بحاجة لتعريف الرؤوس الخاصة بها. ولكن في حالة الأشباح النقطية "Point Sprites"، نحتاج فقط لتعريف رأس واحد لكل شكل. لذا نقوم بالمرور على قائمة الرصاص، و لكل رصاصة نقوم بإضافة رأس من النوع "VertexPointSprite" إلى مصفوفة. هذا الرأس يحمل الموقع الثلاثي الأبعاد و الحجم.

بعدها، نحن بحاجة لإرشاد بطاقة الرسوميات للرسم من هذه الرؤوس باستخدام التقنية "PointsSprites". بما أننا ما زلنا نتعامل مع التحويلات من الثلاثي الأبعاد- إلى الثنائي الأبعاد، سنحتاج إلى كل من مصفوفات العالم، العرض و

الإسقاط. لأن مواقع الرصاص قد تم تعريفه بالنسبة للمحور المطلق (بنفس الطريقة في المدينة ثلاثية الأبعاد، وهو المحور الذي طوله المطلق=1)، لذلك إستخدمنا القيمة المحايدة لمصفوفة العالم لدينا. أيضا قمنا بتحديد صورة الرصاصة باعتبارها الخامة الحالية للرسم، حيث سوف تقوم بطاقة الرسومات بإشتقاق لون كل بكسل منها.

فعليا قد رأينا الجزء الأخير من الكود مرات عديدة: حيث لكل pass في التأثير، نقوم برسم عدد معين من العناصر "Elements" من المصفوفة التي تحتوي على الرؤوس. لأن في حالتنا هذه كل صورة تتطلب رأس واحد فقط، ستكون عدد العناصر التي سوف يتم رسمها مساوي لعدد العناصر الموجودة في المصفوفة spriteArray. لاحظ أن هذا هو المكان الذي نستخدم فيه تعريف الرؤوس "VertexDeclaration" الخاص بنا.

نحن بحاجة إلى وضع القيمة true للخاصية RenderState.PointSpriteEnable، لأن بعض بطاقات الرسومات (مثل المستعملة في ال Xbox360) بحاجة لأن تعرف ذلك. أيضا لا تنسى أن تعطيهما القيمة false في نهاية الدالة.

في هذا المثال، المعالج المركزي "CPU" يقوم بحساب المواقع الجديدة لكل رصاصة في كل إطار في اللعبة، ومن ثم إنشاء مصفوفة رؤوس من المواقع الموجودة، و من ثم إرسالها إلى بطاقة الرسومات في كل إطار. سيكون من الأفضل كثيرا أن نجعل معالج بطاقة الرسومات "GPU" يقوم بهذه العملية. من الممكن إستخدام تقنيات أنظمة الجزيئات لذلك.

بهذا نكون قد أنهينا الدالة ☺، كل ما علينا عمله هو إستدعائها من نهاية الدالة Draw():

```
DrawBullets();
```

الآن حاول تشغيل الكود!

يجب أن ترى شيئا شبيها بالصورة التالية:



- سوف نقوم في الدرس القادم بتحسين شكل الرصاصات بإذن الله. سيتضمن الدرس القادم أيضا عملية إكتشاف تصادم الرصاص بعناصر اللعبة. في حالة وجود تصادم، سوف يتم حذف الرصاصة من قائمة الرصاص bulletList، ما سوف يوفر مساحة في الذاكرة.

حاول حل التمرين التالي:

- قم بتغيير حجم و سرعة الأشباح النقطية "Point Sprites" (الرصاص).

الكود حتى اللحظة:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace XNAseries2
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        struct Bullet
        {
            public Vector3 position;
            public Quaternion rotation;
        }

        private struct VertexPointSprite
        {
            private Vector3 position;
            private float pointSize;

            public VertexPointSprite(Vector3 position, float pointSize)
            {
                this.position = position;
                this.pointSize = pointSize;
            }

            public static readonly VertexElement[] VertexElements =
            {
                new VertexElement(0, 0, VertexElementFormat.Vector3, VertexElementMethod.Default,
                VertexElementUsage.Position, 0),
                new VertexElement(0, sizeof(float)*3, VertexElementFormat.Single,
                VertexElementMethod.Default, VertexElementUsage.PointSize, 0),
            };
            public static int SizeInBytes = sizeof(float) * (3 + 1);
        }

        enum CollisionType { None, Building, Boundary, Target }

        const int maxTargets = 50;
        int[] buildingHeights = new int[] { 0, 2, 2, 6, 5, 4 };

        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Vector3 lightDirection = new Vector3(3, -2, 5);
        Matrix viewMatrix;
        Matrix projectionMatrix;

        Texture2D sceneryTexture;
        Texture2D bulletTexture;
        Model xwingModel;
        Model targetModel;
    }
}
```

```

Vector3 xwingPosition = new Vector3(8, 1, -3);
Quaternion xwingRotation = Quaternion.Identity;

int[,] floorPlan;
float gameSpeed = 1.0f;

VertexBuffer cityVertexBuffer;
VertexDeclaration texturedVertexDeclaration;
VertexDeclaration pointSpriteVertexDeclaration;

BoundingBox[] buildingBoundingBoxes;
BoundingBox completeCityBox;

List<BoundingSphere> targetList = new List<BoundingSphere> ();

List<Bullet> bulletList = new List<Bullet> ();          double lastBulletTime = 0;

public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}

protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "Riemer's XNA Tutorials -- Series 2";

    LoadFloorPlan();
    lightDirection.Normalize();
    SetUpBoundingBoxes();
    AddTargets();

    base.Initialize();
}

protected override void LoadContent()
{
    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect> ("effects");
    sceneryTexture = Content.Load<Texture2D> ("texturemap");
    bulletTexture = Content.Load<Texture2D> ("bullet");
    xwingModel = LoadModel("xwing");
    targetModel = LoadModel("target");

    SetUpVertices();
    SetUpCamera();

    pointSpriteVertexDeclaration = new VertexDeclaration(device,
VertexPointSprite.VertexElements);
}

private Model LoadModel(string assetName)
{
    Model newModel = Content.Load<Model> (assetName);          foreach (ModelMesh mesh in
newModel.Meshes)
        foreach (ModelMeshPart meshPart in mesh.MeshParts)
            meshPart.Effect = effect.Clone(device);
    return newModel;
}

private void LoadFloorPlan()
{
    floorPlan = new int[,]
    {
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,1,1,0,0,0,1,1,0,0,1,0,1},
        {1,0,0,1,1,0,0,0,1,0,0,0,1,0,1},
        {1,0,0,0,1,1,0,1,1,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
    }
}

```

```

        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1},
        {1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
        {1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1},
        {1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
    };

    Random random = new Random();
    int differentBuildings = buildingHeights.Length - 1;
    for (int x = 0; x < floorPlan.GetLength(0); x++)
        for (int y = 0; y < floorPlan.GetLength(1); y++)
            if (floorPlan[x, y] == 1)
                floorPlan[x, y] = random.Next(differentBuildings) + 1;
}

private void SetUpBoundingBoxes()
{
    int cityWidth = floorPlan.GetLength(0);
    int cityLength = floorPlan.GetLength(1);

    List<BoundingBox> bbList = new List<BoundingBox> ();
    for (int x = 0; x <
cityWidth; x++)
    {
        for (int z = 0; z < cityLength; z++)
        {
            int buildingType = floorPlan[x, z];
            if (buildingType != 0)
            {
                int buildingHeight = buildingHeights[buildingType];
                Vector3[] buildingPoints = new Vector3[2];
                buildingPoints[0] = new Vector3(x, 0, -z);
                buildingPoints[1] = new Vector3(x + 1, buildingHeight, -z - 1);
                BoundingBox buildingBox = BoundingBox.CreateFromPoints(buildingPoints);
                bbList.Add(buildingBox);
            }
        }
    }
    buildingBoundingBoxes = bbList.ToArray();

    Vector3[] boundaryPoints = new Vector3[2];
    boundaryPoints[0] = new Vector3(0, 0, 0);
    boundaryPoints[1] = new Vector3(cityWidth, 20, -cityLength);
    completeCityBox = BoundingBox.CreateFromPoints(boundaryPoints);
}

private void AddTargets()
{
    int cityWidth = floorPlan.GetLength(0);
    int cityLength = floorPlan.GetLength(1);

    Random random = new Random();

    while (targetList.Count < maxTargets)
    {
        int x = random.Next(cityWidth);
        int z = -random.Next(cityLength);
        float y = (float)random.Next(2000) / 1000f + 1;
        float radius = (float)random.Next(1000) / 1000f * 0.2f + 0.01f;

        BoundingBox newTarget = new BoundingBox(new Vector3(x, y, z), radius);

        if (CheckCollision(newTarget) == CollisionType.None)
            targetList.Add(newTarget);
    }
}

private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(20, 13, -5), new Vector3(8, 0, -7), new
Vector3(0, 1, 0));
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 0.2f, 500.0f);
}

```

```
}

private void SetUpVertices()
{
    int differentBuildings = buildingHeights.Length - 1;
    float imagesInTexture = 1 + differentBuildings * 2;

    int cityWidth = floorPlan.GetLength(0);
    int cityLength = floorPlan.GetLength(1);

    List<VertexPositionNormalTexture> verticesList = new List<VertexPositionNormalTexture> ();
    for (int x = 0; x < cityWidth; x++)
    {
        for (int z = 0; z < cityLength; z++)
        {
            int currentbuilding = floorPlan[x, z];

            //floor or ceiling
            verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
            buildingHeights[currentbuilding], -z), new Vector3(0, 1, 0), new Vector2((currentbuilding * 2 /
            imagesInTexture, 1))));
            verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
            buildingHeights[currentbuilding], -z - 1), new Vector3(0, 1, 0), new Vector2((currentbuilding * 2) /
            imagesInTexture, 0))));
            verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
            buildingHeights[currentbuilding], -z), new Vector3(0, 1, 0), new Vector2((currentbuilding * 2 + 1) /
            imagesInTexture, 1))));
            verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
            buildingHeights[currentbuilding], -z - 1), new Vector3(0, 1, 0), new Vector2((currentbuilding * 2) /
            imagesInTexture, 0))));
            verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
            buildingHeights[currentbuilding], -z - 1), new Vector3(0, 1, 0), new Vector2((currentbuilding * 2 + 1)
            / imagesInTexture, 0))));
            verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
            buildingHeights[currentbuilding], -z), new Vector3(0, 1, 0), new Vector2((currentbuilding * 2 + 1) /
            imagesInTexture, 1))));

            if (currentbuilding != 0)
            {
                //front wall
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z - 1),
                new Vector3(0, 0, -1), new Vector2((currentbuilding * 2) / imagesInTexture, 1))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
                buildingHeights[currentbuilding], -z - 1), new Vector3(0, 0, -1), new Vector2((currentbuilding * 2 - 1)
                / imagesInTexture, 0))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x, 0, -z - 1), new
                Vector3(0, 0, -1), new Vector2((currentbuilding * 2 - 1) / imagesInTexture, 1))));

                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
                buildingHeights[currentbuilding], -z - 1), new Vector3(0, 0, -1), new Vector2((currentbuilding * 2 - 1)
                / imagesInTexture, 0))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z - 1),
                new Vector3(0, 0, -1), new Vector2((currentbuilding * 2) / imagesInTexture, 1))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
                buildingHeights[currentbuilding], -z - 1), new Vector3(0, 0, -1), new Vector2((currentbuilding * 2) /
                imagesInTexture, 0))));

                //back wall
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z), new
                Vector3(0, 0, 1), new Vector2((currentbuilding * 2) / imagesInTexture, 1))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x, 0, -z), new
                Vector3(0, 0, 1), new Vector2((currentbuilding * 2 - 1) / imagesInTexture, 1))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
                buildingHeights[currentbuilding], -z), new Vector3(0, 0, 1), new Vector2((currentbuilding * 2 - 1) /
                imagesInTexture, 0))));

                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
                buildingHeights[currentbuilding], -z), new Vector3(0, 0, 1), new Vector2((currentbuilding * 2 - 1) /
                imagesInTexture, 0))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
                buildingHeights[currentbuilding], -z), new Vector3(0, 0, 1), new Vector2((currentbuilding * 2) /
                imagesInTexture, 0))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z), new
                Vector3(0, 0, 1), new Vector2((currentbuilding * 2) / imagesInTexture, 1))));

                //left wall
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x, 0, -z), new
                Vector3(-1, 0, 0), new Vector2((currentbuilding * 2) / imagesInTexture, 1))));
                verticesList.Add(new VertexPositionNormalTexture(new Vector3(x, 0, -z - 1), new
```

```
Vector3(-1, 0, 0), new Vector2((currentbuilding * 2 - 1) / imagesInTexture, 1));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
buildingHeights[currentbuilding], -z - 1), new Vector3(-1, 0, 0), new Vector2((currentbuilding * 2 - 1)
/ imagesInTexture, 0)));

    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
buildingHeights[currentbuilding], -z - 1), new Vector3(-1, 0, 0), new Vector2((currentbuilding * 2 - 1)
/ imagesInTexture, 0)));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x,
buildingHeights[currentbuilding], -z), new Vector3(-1, 0, 0), new Vector2((currentbuilding * 2) /
imagesInTexture, 0)));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x, 0, -z), new
Vector3(-1, 0, 0), new Vector2((currentbuilding * 2) / imagesInTexture, 1)));

    //right wall
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z), new
Vector3(1, 0, 0), new Vector2((currentbuilding * 2) / imagesInTexture, 1)));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
buildingHeights[currentbuilding], -z - 1), new Vector3(1, 0, 0), new Vector2((currentbuilding * 2 - 1)
/ imagesInTexture, 0)));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z - 1),
new Vector3(1, 0, 0), new Vector2((currentbuilding * 2 - 1) / imagesInTexture, 1)));

    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
buildingHeights[currentbuilding], -z - 1), new Vector3(1, 0, 0), new Vector2((currentbuilding * 2 - 1)
/ imagesInTexture, 0)));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1, 0, -z), new
Vector3(1, 0, 0), new Vector2((currentbuilding * 2) / imagesInTexture, 1)));
    verticesList.Add(new VertexPositionNormalTexture(new Vector3(x + 1,
buildingHeights[currentbuilding], -z), new Vector3(1, 0, 0), new Vector2((currentbuilding * 2) /
imagesInTexture, 0)));
    }
}

cityVertexBuffer = new VertexBuffer(device, verticesList.Count *
VertexPositionNormalTexture.SizeInBytes, BufferUsage.WriteOnly);

cityVertexBuffer.SetData<VertexPositionNormalTexture> (verticesList.ToArray());
texturedVertexDeclaration = new VertexDeclaration(device,
VertexPositionNormalTexture.VertexElements);
}

protected override void UnloadContent()
{
}

protected override void Update(GameTime gameTime)
{
    ProcessKeyboard(gameTime);
    float moveSpeed = gameTime.ElapsedGameTime.Milliseconds / 500.0f * gameSpeed;
    MoveForward(ref xwingPosition, xwingRotation, moveSpeed);

    BoundingBox xwingSpere = new BoundingBox(xwingPosition, 0.04f);
    if (CheckCollision(xwingSpere) != CollisionType.None)
    {
        xwingPosition = new Vector3(8, 1, -3);
        xwingRotation = Quaternion.Identity;
        gameSpeed /= 1.1f;
    }

    UpdateCamera();

    UpdateBulletPositions(moveSpeed);

    base.Update(gameTime);
}

private void UpdateCamera()
{
    Vector3 campos = new Vector3(0, 0.1f, 0.6f);
    campos = Vector3.Transform(campos, Matrix.CreateFromQuaternion(xwingRotation));
    campos += xwingPosition;

    Vector3 camup = new Vector3(0, 1, 0);
    camup = Vector3.Transform(camup, Matrix.CreateFromQuaternion(xwingRotation));

    viewMatrix = Matrix.CreateLookAt(campos, xwingPosition, camup);
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 0.2f, 500.0f);
}
```

```

private void ProcessKeyboard(GameTime gameTime)
{
    float leftRightRot = 0;

    float turningSpeed = (float)gameTime.ElapsedGameTime.TotalMilliseconds / 1000.0f;
    turningSpeed *= 1.6f * gameSpeed;
    KeyboardState keys = Keyboard.GetState();
    if (keys.IsKeyDown(Keys.Right))
        leftRightRot += turningSpeed;
    if (keys.IsKeyDown(Keys.Left))
        leftRightRot -= turningSpeed;

    float upDownRot = 0;
    if (keys.IsKeyDown(Keys.Down))
        upDownRot += turningSpeed;
    if (keys.IsKeyDown(Keys.Up))
        upDownRot -= turningSpeed;

    Quaternion additionalRot = Quaternion.CreateFromAxisAngle(new Vector3(0, 0, -1),
leftRightRot) * Quaternion.CreateFromAxisAngle(new Vector3(1, 0, 0), upDownRot);
    xwingRotation *= additionalRot;

    if (keys.IsKeyDown(Keys.Space))
    {
        double currentTime = gameTime.TotalGameTime.TotalMilliseconds;
        if (currentTime - lastBulletTime > 100)
        {
            Bullet newBullet = new Bullet();
            newBullet.position = xwingPosition;
            newBullet.rotation = xwingRotation;
            bulletList.Add(newBullet);

            lastBulletTime = currentTime;
        }
    }
}

private void MoveForward(ref Vector3 position, Quaternion rotationQuat, float speed)
{
    Vector3 addVector = Vector3.Transform(new Vector3(0, 0, -1), rotationQuat);
    position += addVector * speed;
}

private CollisionType CheckCollision(BoundingSphere sphere)
{
    for (int i = 0; i < buildingBoundingBoxes.Length; i++)
        if (buildingBoundingBoxes[i].Contains(sphere) != ContainmentType.Disjoint)
            return CollisionType.Building;

    if (completeCityBox.Contains(sphere) != ContainmentType.Contains)
        return CollisionType.Boundary;

    for (int i = 0; i < targetList.Count; i++)
    {
        if (targetList[i].Contains(sphere) != ContainmentType.Disjoint)
        {
            targetList.RemoveAt(i);
            i--;
            AddTargets();

            return CollisionType.Target;
        }
    }

    return CollisionType.None;
}

private void UpdateBulletPositions(float moveSpeed)
{
    for (int i = 0; i < bulletList.Count; i++)
    {
        Bullet currentBullet = bulletList[i];
        MoveForward(ref currentBullet.position, currentBullet.rotation, moveSpeed * 2.0f);
        bulletList[i] = currentBullet;
    }
}

protected override void Draw(GameTime gameTime)
{
    device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.DarkSlateBlue, 1.0f,

```

```

0);

    DrawCity();
    DrawModel();
    DrawTargets();
    DrawBullets();

    base.Draw(gameTime);
}

private void DrawCity()
{
    effect.CurrentTechnique = effect.Techniques["Textured"];
    effect.Parameters["xWorld"].SetValue(Matrix.Identity);
    effect.Parameters["xView"].SetValue(viewMatrix);
    effect.Parameters["xProjection"].SetValue(projectionMatrix);
    effect.Parameters["xTexture"].SetValue(sceneryTexture);
    effect.Parameters["xEnableLighting"].SetValue(true);
    effect.Parameters["xLightDirection"].SetValue(lightDirection);
    effect.Parameters["xAmbient"].SetValue(0.5f);
    effect.Begin();
    foreach (EffectPass pass in effect.CurrentTechnique.Passes)
    {
        pass.Begin();
        device.VertexDeclaration = texturedVertexDeclaration;
        device.Vertices[0].SetSource(cityVertexBuffer, 0,
VertexPositionNormalTexture.SizeInBytes);
        device.DrawPrimitives(PrimitiveType.TriangleList, 0, cityVertexBuffer.SizeInBytes /
VertexPositionNormalTexture.SizeInBytes / 3);
        pass.End();
    }
    effect.End();
}

private void DrawModel()
{
    Matrix worldMatrix = Matrix.CreateScale(0.0005f, 0.0005f, 0.0005f) *
Matrix.CreateRotationY(MathHelper.Pi) * Matrix.CreateFromQuaternion(xwingRotation) *
Matrix.CreateTranslation(xwingPosition);

    Matrix[] xwingTransforms = new Matrix[xwingModel.Bones.Count];
    xwingModel.CopyAbsoluteBoneTransformsTo(xwingTransforms);
    foreach (ModelMesh mesh in xwingModel.Meshes)
    {
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques["Colored"];
            currentEffect.Parameters["xWorld"].SetValue(xwingTransforms[mesh.ParentBone.Index]
* worldMatrix);
            currentEffect.Parameters["xView"].SetValue(viewMatrix);
            currentEffect.Parameters["xProjection"].SetValue(projectionMatrix);
            currentEffect.Parameters["xEnableLighting"].SetValue(true);
            currentEffect.Parameters["xLightDirection"].SetValue(lightDirection);
            currentEffect.Parameters["xAmbient"].SetValue(0.5f);
        }
        mesh.Draw();
    }
}

private void DrawTargets()
{
    for (int i = 0; i < targetList.Count; i++)
    {
        Matrix worldMatrix = Matrix.CreateScale(targetList[i].Radius) *
Matrix.CreateTranslation(targetList[i].Center);

        Matrix[] targetTransforms = new Matrix[targetModel.Bones.Count];
        targetModel.CopyAbsoluteBoneTransformsTo(targetTransforms);
        foreach (ModelMesh mesh in targetModel.Meshes)
        {
            foreach (Effect currentEffect in mesh.Effects)
            {
                currentEffect.CurrentTechnique = currentEffect.Techniques["Colored"];
                currentEffect.Parameters["xWorld"].SetValue(targetTransforms[mesh.ParentBone.I
ndex] * worldMatrix);
                currentEffect.Parameters["xView"].SetValue(viewMatrix);
                currentEffect.Parameters["xProjection"].SetValue(projectionMatrix);
                currentEffect.Parameters["xEnableLighting"].SetValue(true);
                currentEffect.Parameters["xLightDirection"].SetValue(lightDirection);
                currentEffect.Parameters["xAmbient"].SetValue(0.5f);
            }
        }
    }
}

```

```
        mesh.Draw();
    }
}

private void DrawBullets()
{
    if (bulletList.Count > 0)
    {
        VertexPointSprite[] spriteArray = new VertexPointSprite[bulletList.Count];
        for (int i = 0; i < bulletList.Count; i++)
            spriteArray[i] = new VertexPointSprite(bulletList[i].position, 50);

        effect.CurrentTechnique = effect.Techniques["PointSprites"];
        Matrix worldMatrix = Matrix.Identity;
        effect.Parameters["xWorld"].SetValue(worldMatrix);
        effect.Parameters["xView"].SetValue(viewMatrix);
        effect.Parameters["xProjection"].SetValue(projectionMatrix);
        effect.Parameters["xTexture"].SetValue(bulletTexture);

        device.RenderState.PointSpriteEnable = true;

        effect.Begin();
        foreach (EffectPass pass in effect.CurrentTechnique.Passes)
        {
            pass.Begin();
            device.VertexDeclaration = pointSpriteVertexDeclaration;
            device.DrawUserPrimitives(PrimitiveType.PointList, spriteArray, 0,
spriteArray.Length);
            pass.End();
        }
        effect.End();

        device.RenderState.PointSpriteEnable = false;
    }
}
}
```