Introduction to XNA

2D Shooter game – Part 2
2D Shooter game
Riemer’s XNA tutorials

- You are allowed to progress ahead of me by reading and doing to tutorial yourself.
- I’ll be explaining the tutorial and will be taking things a bit slow.
Run the game – Where were we?

- Cannon angle: 28
- Cannon power: 280
Defining the colors of a texture per-pixel

- Now the rocket is going around the place but it is not hitting the ground.
- To model hitting the ground, we should know exactly where the ground pixels are.
- So let’s generate the ground instead of loading it from an image.
First we need an array to keep the contour data of the ground.

```java
int[] terrainContour;
```

For now let’s create a flat land.

```java
private void GenerateTerrainContour()
{
    terrainContour = new int[screenWidth];

    for (int x = 0; x < screenWidth; x++)
        terrainContour[x] = screenHeight / 2;
}
```
Defining the colors of a texture per-pixel

- We have the upper limit of the land in Y-coordinates now. Next we need to fill up the pixels using that knowledge and the X-coordinates.
- The first line of CreateForeground method creates an array to keep the colour data of all the screen pixels.
- Add the lines to create the texture and set the pixel data.
Random terrain generation

- Alter the code of the `GenerateTerrainContour` method as follows:

```csharp
private void GenerateTerrainContour()
{
    terrainContour = new int[screenWidth];

    float offset = screenHeight / 2;
    float peakheight = 100;
    float flatness = 50;
    for (int x = 0; x < screenWidth; x++)
    {
        double height = peakheight * Math.Sin((float)x / flatness)+offset;
        terrainContour[x] = (int)height;
    }
}
```

- Run the game and see.
Random terrain generation

- **Offset** – simply sets the position of the midheight of the wave.
- **Peakheight** – is multiplied by the output of the Sine function, so it defines how high the wave will be.
- **Flatness** – has an impact on the influence of X, which slows down or speeds up the wave. This increases or decreases the wavelength of our wave.
Random terrain generation

- It looked too artificial.
- We can fix it by adding a second wave.
- And a third!
- To make the terrain be random, we will make the three controlling variables to be random.
Run the game

Cannon angle: 90
Cannon power: 100
Random terrain generation

- Now what is wrong?
  - The players are not on the ground.
- Fix this by altering the Y positions of the players where we generate them.
- And let’s fix the X position so that they are placed in an equal distance.
- For this to work the `GenerateTerrainContour` method call should come before the `SetUpPlayers` method call.
Run the game
Random terrain generation

- Now the origin of the cannon is on the ground.
- But still the cannons themselves are kind of off the ground!
- To fix this we will flatten the ground below the players.
Run the game

Cannon angle: 90
Cannon power: 100
Now the ground looks ok but a plain green ground is boring.

Let's add some texture to the ground.

Download the texture file from [here](#).

Create the `groundTexture` variable and load it to the game.
First create the method to copy texture data to a colour array.

```csharp
private Color[,] TextureTo2DArray(Texture2D texture)
{
    Color[,] colors2D = new Color[texture.Width, texture.Height];
    for (int x = 0; x < texture.Width; x++)
        for (int y = 0; y < texture.Height; y++)
            colors2D[x, y] = colors1D[x + y * texture.Width];

    return colors2D;
}
```
Now go to the `CreateForeground` method and add the following line at the top.

```
Color[,] groundColors = TextureTo2DArray(groundTexture);
```

And replace the foreground colour setting line with the following line.

```
foregroundColors[x + y * screenWidth] = groundColors[x, y];
```

This is OK only if the screen is smaller than the ground image. So we’ll wrap the image around.

```
foregroundColors[x + y * screenWidth] = groundColors[x % groundTexture.Width, y % groundTexture.Height];
```
Run the game

Cannon angle: 31
Cannon power: 100
An image is always a rectangle. Since very few objects are really rectangular, most images contain a lot of transparent pixels. This is shown in the left image below. Note that the transparent pixels have a greenish colour, while the non-transparent pixels have a soft-red colour.
Let’s say both images are positioned so the rocket is positioned in front of the cannon, much like shown in the left part of the image below.

Although in the left part the rectangles of both images collide, we see the cannon and rocket actually don’t collide. This is because in all pairs of overlapping pixels, at least one of both is transparent.
When comparing 2 colliding pixels, if either of them is transparent, there is no collision between the actually object. Only when both the pixel of the first and of the second image are non-transparent, there is a real collision. This is presented in the table below:

<table>
<thead>
<tr>
<th>Pixel Image 1 Transparent?</th>
<th>Pixel Image 2 Transparent?</th>
<th>Collision?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
2D Collision Detection

- When our rocket collides with a cannon, it will both be scaled and rotated. This situation is shown in the left part of the image below.
- The right part if the image above shows the most complicated case: where both images are scaled and/or rotated. We will meet this case in our game.
We will create a method, `TexturesCollide`, which will return whether 2 images collide. The method will need 4 things in order to work properly:
- The 2D array of colors of image 1
- The matrix of image 1
- The 2D array of colors of image 2
- The matrix of image 2

The method will not only return whether there is a collision, but if there is one it will return the screen position of where the collision occurred. If no collision was detected, \((-1,-1)\) will be returned.
private Vector2 TexturesCollide(Color[,] tex1, Matrix mat1, Color[,] tex2, Matrix mat2)
{
    return new Vector2(-1, -1);
}

› Now we need to model “For each pixel in image A”

    int width1 = tex1.GetLength(0);
    int height1 = tex1.GetLength(1);
    for (int x1 = 0; x1 < width1; x1++)
    {
        for (int y1 = 0; y1 < height1; y1++)
        {
"}
2D Collision Detection

For each pixel of image 1, we first want to find the corresponding screen coordinate. This is done by transforming the original X,Y coordinate with the matrix of image 1, so add these lines inside the double for-loop:

```csharp
Vector2 pos1 = new Vector2(x1,y1);
Vector2 screenCoord = Vector2.Transform(pos1, mat1);
```
Now we have the screen coordinate of the current pixel, let’s find to which pixel this correspond in the original image 2. This is the opposite of what we’ve done previously, so now we need to transform the screen coordiante by the *inverse* of the matrix of image 2:

Matrix inverseMat2 = Matrix.Invert(mat2);
Vector2 pos2 = Vector2.Transform(screenCoord, inverseMat2);
2D Collision Detection

- we have 2 positions in the original images, of which we know they are drawn to the same screen pixel. All we need to do is check whether they are both non-transparent:

```cpp
if (tex1[x1, y1].A > 0)
{
    if (tex2[x2, y2].A > 0)
    {
        return screenCoord;
    }
}
```
2D Collision Detection

- This implements the core functionality of the method. While there might be cases that this method survives, in general it will crash, because some pixels of image A will fall outside of image B. In the image above for example: the very first pixel the rocket (to the left of the top of the rocket) does not correspond to a pixel in image B.
2D Collision Detection

- This is easily solvable, by checking whether the coordinate for image B is not outside image B:

```csharp
int width2 = tex2.GetLength(0);
int height2 = tex2.GetLength(1);
int x2 = (int)pos2.X;
int y2 = (int)pos2.Y;
if ((x2 >= 0) && (x2 < width2))
{
    if ((y2 >= 0) && (y2 < height2))
    {
        if (tex1[x1, y1].A > 0)
        {
            if (tex2[x2, y2].A > 0)
            {
                return screenCoord;
            }
        }
    }
}
```
2D Collision Detection

- In our method, for each pixel of image A we’re doing two transformations: from image A to screen coordinates, and from these screen coordinates to image B. It would be better to replace them by a single transformation: from image A to image B.

- Using matrices, this is rather easy. Now, we’re transforming a coordinate by mat1, and then by inverse(mat2). We can obtain the matrix that is the combination of both, simply by multiplying them. Put this line at the top of the method:

  ```
  Matrix mat1to2 = mat1 * Matrix.Invert(mat2);
  ```
Finally the method looks like this:

```csharp
private Vector2 TexturesCollide(Color[,] tex1, Matrix mat1, Color[,] tex2, Matrix mat2)
{
    Matrix mat1to2 = mat1 * Matrix.Invert(mat2);
    int width1 = tex1.GetLength(0);
    int height1 = tex1.GetLength(1);
    int width2 = tex2.GetLength(0);
    int height2 = tex2.GetLength(1);

    for (int x1 = 0; x1 < width1; x1++)
    {
        for (int y1 = 0; y1 < height1; y1++)
        {
            Vector2 pos1 = new Vector2(x1, y1);
            Vector2 pos2 = Vector2.Transform(pos1, mat1to2);
            int x2 = (int)pos2.X;
            int y2 = (int)pos2.Y;
            if ((x2 >= 0) && (x2 < width2))
            {
                if ((y2 >= 0) && (y2 < height2))
                {
                    if (tex1[x1, y1].A > 0)
                    {
                        if (tex2[x2, y2].A > 0)
                        {
                            Vector2 screenPos = Vector2.Transform(pos1, mat1);
                            return screenPos;
                        }
                    }
                }
            }
        }
    }
    return new Vector2(-1, -1);
}
```
2D Transformation matrices

- Start by adding these variable to the top of our code:

  ```
  Color[,] rocketColorArray;
  Color[,] foregroundColorArray;
  Color[,] carriageColorArray;
  Color[,] cannonColorArray;
  ```

- Foreground is initialized at the `CreateForeground` method. Other three is initialized at the `LoadContent` method.
We did not change (transform) the foreground image.
Therefore the transformation matrix of our foreground image is?
- The Identity matrix!
- Matrix foregroundMat = Matrix.Identity;
2D Transformation matrices

- Remember, order of multiplication is important in Matrices.
- So let’s see what are the steps that XNA take to render an image on the screen.
2D Transformation matrices

- If we would render the image just like it is (or: with the Identity transformation), it would be rendered in its original size in the top-left corner of the screen.
2D Transformation matrices

- First, the carriage image is moved so its top-left point is at the position specified as second argument in the SpriteBatch.Draw method.
Then, everything is scaled down. You see this includes the images, as well as its own X and Y axis.
Finally, and this is the most challenging step: the image is moved over the Y axis, since in our SpriteBatch.Draw method we’ve specified (0, carriageTexture.Height) as origin.
2D Transformation matrices

- 1: Matrix.Identity;
- 2: Matrix.CreateTranslation(xPos, yPos, 0)
- 3: Matrix.CreateScale(playerScaling)
- 4: Matrix.CreateTranslation(0, –carriage.Height, 0)

Matrix carriageMat = Matrix.CreateTranslation(0, –carriage.Height, 0) * Matrix.CreateScale(playerScaling) * Matrix.CreateTranslation(xPos, yPos, 0) * Matrix.Identity;
2D Transformation matrices

- There was no rotation.
- What would happen if we needed to rotate?
- It comes between Scaling and moving the origin.
2D Transformation matrices
Matrix rocketMat =
Matrix.CreateTranslation(-42, -240, 0) *
Matrix.CreateRotationZ(rocketAngle) *
Matrix.CreateScale(rocketScaling) *
Matrix.CreateTranslation(rocketPosition.X, rocketPosition.Y, 0);
2D Transformation matrices

Finally we have:

```
Matrix foregroundMat = Matrix.Identity;

Matrix carriageMat =
    Matrix.CreateTranslation(0, -carriage.Height, 0) *
    Matrix.CreateScale(playerScaling) *
    Matrix.CreateTranslation(xPos, yPos, 0) *
    Matrix.Identity;

Matrix rocketMat =
    Matrix.CreateTranslation(-42, -240, 0) *
    Matrix.CreateRotationZ(rocketAngle) *
    Matrix.CreateScale(rocketScaling) *
    Matrix.CreateTranslation(rocketPosition.X, rocketPosition.Y, 0);

Matrix cannonMat =
    Matrix.CreateTranslation(cannonOrigin) *
    Matrix.CreateRotationZ(player.Angle) *
    Matrix.CreateScale(playerScaling) *
    Matrix.CreateTranslation(xPos + 20, yPos - 10, 0);
```
First the collision between the rocket and the terrain.

```csharp
private Vector2 CheckTerrainCollision()
{
    Matrix rocketMat = Matrix.CreateTranslation(-42, -240, 0) *
    Matrix.CreateRotationZ(rocketAngle) * Matrix.CreateScale(rocketScaling) *
    Matrix.CreateTranslation(rocketPosition.X, rocketPosition.Y, 0);

    Matrix terrainMat = Matrix.Identity;

    Vector2 terrainCollisionPoint =TexturesCollide(rocketColorArray,
    rocketMat, foregroundColorArray, terrainMat);

    return terrainCollisionPoint;
}
```
Now the `CheckPlayersCollision` method, which promises to be a little more complex as there are multiple players, and some of them might no longer be alive.

The matrices of the rocket and carriage are passed to the `TexturesCollide` method, and the result is stored in a the `carriageCollisionPoint` Vector2. Remember that this contains (−1,−1) if no collision was detected. So if it has collided, we have to set it as dead.
Putting Collision Detection into practice

- If a collision between the rocket and the current carriage is detected, the IsAlive property of the colliding player is set to false and the method returns the collision point, which immediately terminates the method.

```csharp
if (carriageCollisionPoint.X > -1)
{
    players[i].IsAlive = false;
    return carriageCollisionPoint;
}
```
If no collision was detected, we should check for collisions between the rocket and the cannon of the current player. This is done exactly the same, only this time we create the matrix for the cannon, instead of for the carriage: (put this code after the previous lines)

```csharp
Matrix cannonMat = Matrix.CreateTranslation(-11, -50, 0) * Matrix.CreateRotationZ(player.Angle) * Matrix.CreateScale(playerScaling) * Matrix.CreateTranslation(xPos + 20, yPos - 10, 0);
Vector2 cannonCollisionPoint = TexturesCollide(cannonColorArray, cannonMat, rocketColorArray, rocketMat);
if (cannonCollisionPoint.X > -1)
{
    players[i].IsAlive = false;
    return cannonCollisionPoint;
}
```
Finally, we need a method to check whether the rocket is still inside the window. This shouldn’t be that difficult:

```csharp
private bool CheckOutOfScreen()
{
    bool rocketOutOfScreen = rocketPosition.Y > screenHeight;
    rocketOutOfScreen |= rocketPosition.X < 0;
    rocketOutOfScreen |= rocketPosition.X > screenWidth;

    return rocketOutOfScreen;
}
```
Now create a general method that processes their results:

```csharp
private void CheckCollisions(GameTime gameTime)
{
    Vector2 terrainCollisionPoint = CheckTerrainCollision();
    Vector2 playerCollisionPoint = CheckPlayersCollision();
    bool rocketOutOfScreen = CheckOutOfScreen();

    if (playerCollisionPoint.X > -1)
    {
        rocketFlying = false;
        smokeList = new List<Vector2>();
        NextPlayer();
    }

    if (terrainCollisionPoint.X > -1)
    {
        rocketFlying = false;
        smokeList = new List<Vector2>();
        NextPlayer();
    }

    if (rocketOutOfScreen)
    {
        rocketFlying = false;
        smokeList = new List<Vector2>();
        NextPlayer();
    }
}
```
Putting Collision Detection into practice

- Now we need to change the Update method.

```csharp
if (rocketFlying)
{
    UpdateRocket();
    CheckCollisions(gameTime);
}
```

- And add a next player method.

```csharp
private void NextPlayer()
{
    currentPlayer = currentPlayer + 1;
    currentPlayer = currentPlayer % numberOfPlayers;
    while (!players[currentPlayer].IsAlive)
    {
        currentPlayer = ++currentPlayer % numberOfPlayers;
    }
}
```
Run the game and kill others!

Cannon angle: 39
Cannon power: 626
Explosions!

- First download the particle texture by clicking [here](#).
- Import, Create the variable and load it.
- Just like for the players, we need a struct.
- And just like the smoke, we need a list.
- Now create `AddExplosion` method and `AddExplosionParticle` method.
- Add the randomization calculations to the `AddExplosionParticle` method.
- Now add the created particle in to the particle list.
Explosions!

- Add explosion method calls to the `CheckCollisions` method.
- Add the `DrawExplosion` method.
  - See how each of the particles are rotated.
- Finally call the `DrawExplosion` method from the `Draw` method.
Run the game

Cannon angle: 83
Cannon power: 100
Additive alpha blending

- We need to get rid of the black backgrounds.
- you are interested in obtaining the new color for a pixel, given the following 2 colors:
  - The **SourceColor**: The new color, coming from the image that is currently being drawn.
  - The **DestinationColor**: The color for that pixel that was already present in the backbuffer and is about to be overwritten.
- \( \text{NewColor} = \text{SrcBlend} \times \text{SrcColor} + \text{DestBlend} \times \text{DestColor} \)
Additive alpha blending

- Alpha values?
- Default alpha blending in XNA the SpriteBatch uses this Alpha value as SrcBlend factor. The inverse value, \((1 - \text{SrcBlend})\) is used as DestBlend:
- Black is \((0,0,0)\) So we’ll just set each factor to 1.
  - \(\text{NewColor} = 1 \times \text{SrcColor} + 1 \times \text{DestColor}\)
  - \(\text{NewColor} = \text{SrcColor} + \text{DestColor}\)
Additive alpha blending

- BlendState.Additive ← To use additive blending.
- SpriteSortMode.Deferred ← So that only the explosion is drawn using additive blending.
Run the game

Cannon angle: 80
Cannon power: 100
At this moment, whenever the rocket hits the terrain or a player, some particles are created, added to a list and rendered to the screen using additive alpha blending. However, the particles aren’t moving yet.

Based on the initial position, the initial speed and the acceleration you can find the current position.

$$\text{pos}_{\text{current}} = \frac{1}{2} \cdot \text{acceleration} \cdot \text{time}^2 + \text{speed}_{\text{initial}} \cdot \text{time} + \text{pos}_{\text{initial}}$$

Looks familiar?
Add the `UpdateParticles` method.
Calculate relative age.
Model the equation from the previous slide.
Add the Modulation colour variable to add the fading effect.
Update the `DrawExplosion` method
Add code line to make the particle grow as it near its end.
Update the entry in the `particleList`
Particle engine

- Call the **UpdateParticles** method from the **Update** method.
- Run the game.
  - What is wrong?
- Let’s add 2 constraints.
  - The final speed to be exactly 0
  - The final position should be the starting position + the “displacement” vector calculated in our **AddExplosionParticle**
Let’s add 2 constraints.

- The final speed to be exactly 0
- The final position should be the starting position + the “displacement” vector calculated in our AddExplosionParticle

\[
\text{speed}_{\text{final}} = 0 = \text{speed}_{\text{initial}} + \text{acceleration} \times 1
\]

\[
\text{acceleration} = -\text{speed}_{\text{initial}}
\]
Let’s add 2 constraints.

- The final speed to be exactly 0
- The final position should be the starting position + the “displacement” vector calculated in our `AddExplosionParticle`.

\[
\text{pos}_{\text{initial}} + \text{displacement} = \frac{1}{2} \cdot \text{acceleration} \cdot t^2 + \text{speed}_{\text{initial}} \cdot t + \text{pos}_{\text{initial}}
\]

\[
\text{displacement} = \frac{1}{2} \cdot (-\text{speed}_{\text{initial}}) + \text{speed}_{\text{initial}} = \frac{1}{2} \cdot \text{speed}_{\text{initial}}
\]

\[
\text{speed}_{\text{initial}} = 2 \cdot \text{displacement}
\]
Particle engine

- Update the `particle.Direction` variable's and the `particle.Acceleration` variable's values.
Run the game

Cannon angle: 15
Cannon power: 340
Adding explosion craters

- Create the `AddCrater` method.
- Make sure the screen coordinate is not outside the screen.
- Create the `explosionColorArray` variable to store the explosion color array.
- Add a random rotation.
- Construct the Matrix for the explosion image.
- Call the `AddCrater` method.
- Add calls to update the player position and recreate the foreground.
Run the game

Cannon angle: 17
Cannon power: 380
Do the final two additions at home.
Riemer’s XNA Tutorials

- Four series – terrain, flight simulator, higher-level shader language, advanced terrain. I did 2, 1, and 3 in that order.
- Great intro to 3D games using XNA
- He answers questions.
Riemer’s XNA Tutorials
XNA Books

Professional XNA Game Programming for Xbox 360 and Windows
Benjamin Nitschke
Wrox (Wiley) 2007

He created Rocket Commander XNA and Racing Game, both available for Windows and Xbox 360.

Other books to appear.
Creators Club Online

XNA Team Site

- http://creators.xna.com/
- Download XNA, Games
- Tutorials, Code examples
References

- A Games Class Using XNA Game Studio Express (http://www.cecs.csulb.edu/~artg/games.ppt) by Prof. Art Gittleman, California State University Long Beach
- 3D Game Programming using XNA GSE (http://www.slideshare.net/guest3860287/xna-demoppt) by Sahithya Baskaran, PES Institute Of Technology
- 2D Graphics in XNA Game Studio Express (http://classes.soe.ucsc.edu/cmps020/Winter08/lectures/intro-xna-gse-2d.ppt) by Prof. Jim Whitehead