-

Friday, May 18, 2007 1:47 PM by dawate

Let's Make a Game with XNA

DOWNLOAD THE SOURCE HERE! (~44MB)

I admit that I have a deep-seated love for PC and console games. Naturally, as someone with a programmer's background, I have tried making my own games on different platforms since the day I first #included stdio.h. My attempts (all of which failed) progressed over the years across most languages we see games written in today: C, C++, Java, C#, Flash, VB, and so forth. I always got waylaid by something, though, and it frustrated me.

With the more "hardcore" (so-to-speak) unmanaged languages like C and C++, I learned a few things very quickly. You have to know a lot about matrices and 3D math, and come up with a way to represent that using arcane C/C++ constructs. To make code more performant, you'd see a lot of inline assembly. I found that hacking out something this technical meant that you had a ton of engine work to do **before** you could start doing creative things. As an enthusiast, I didn't want that kind of long-term commitment. So I gave up on that.

With the interpreted platforms like Java and Flash, I realized I was working on platforms with very limited power. If I was going to get into game development, I wanted a safe, easy place to start and a lot of room to grow. So I built some simple things for the Web and again I gave up.

Now we have XNA. With XNA and this extremely limited knowledge of how games work, I was able to build a simple, functional game based on 2D sprites in about 10 hours (including the [less-than-impressive] art and music) with zero knowledge of XNA per se. And although XNA is based on the managed .NET Framework, it is optimized for DirectX and is much different/faster/easier than trying to write a game that calls into GDI/GDI+.

Enough with the background! Let's get started making a really really simple game based on 2D sprites. A rehash of an old classic. Without any further ado, I present to you:

IMPORTAMT! (*I'm Moving Paddles Onscreen Relative To A Moving Target!*)

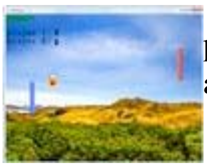
The solution is actually called XNAPong, but that's not importamt...

Article Contents

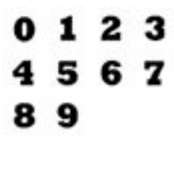
See the next article (TBD) for adding audio with XACT.

What'll it Look Like?

Okay, my eager minions, I'll throw some screenshots up. Behold the glory of 2D sprites that I created in Paint.NET.


I realize this is not the most fantastic game interface ever created. I would like to harken back to the "10 hours" comment from earlier in the article. The background art a stock desktop background from Windows.

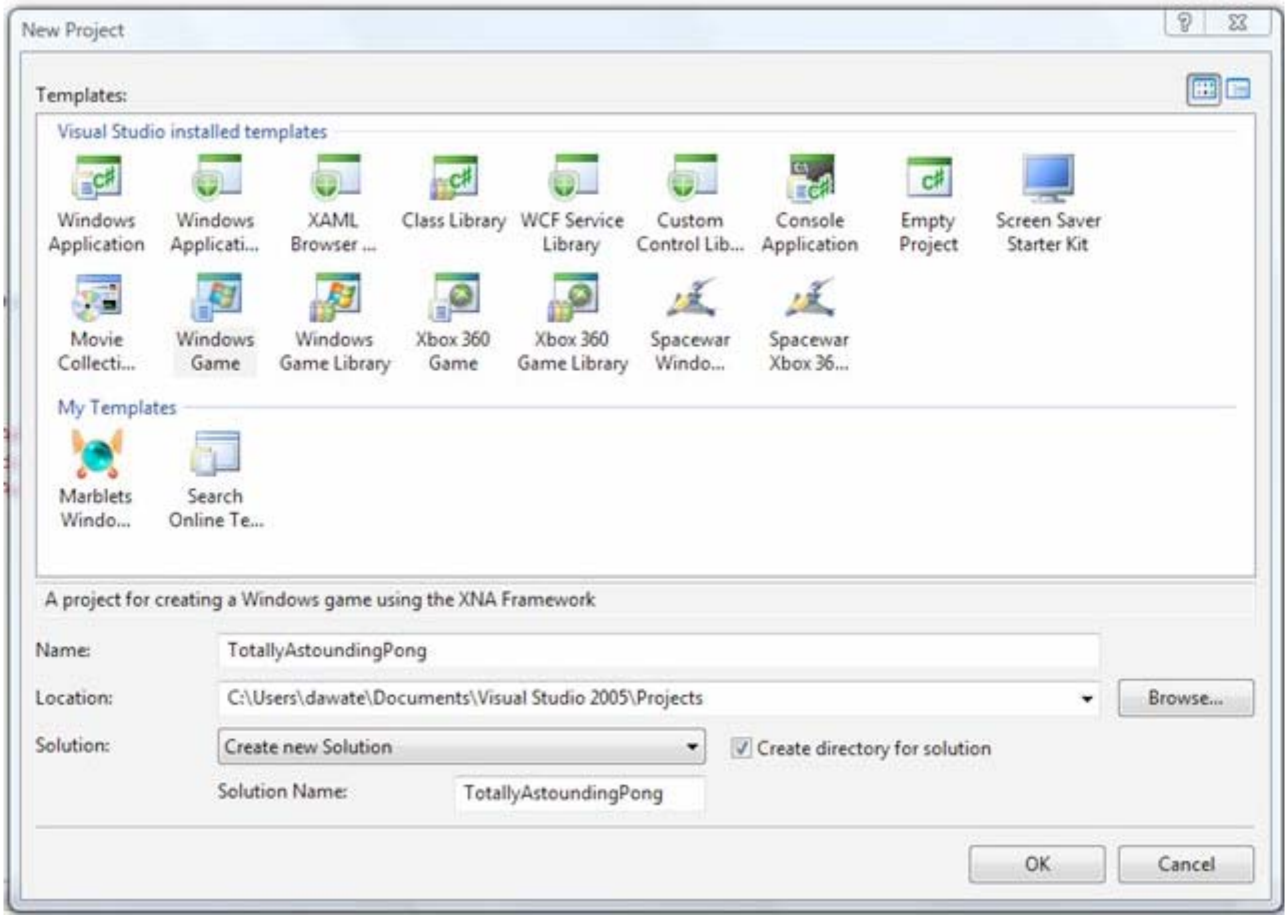
Pongoid himself, in all his glory. A reluctant hero who has resigned himself to a fate of being smacked around for the entire lifespan of the game. Notice that he has two, extremely detailed poses: Screaming While Traveling Left, and Screaming While Traveling Right. In more complicated animations, we'd have actual frames here, and flip through them based on time passed (the entire sheet of frames is loaded as one XNA resource when it comes to sprites).



To give you an idea of what an amazing programmer I am, to your left is the NumberMap. This is how I displayed the score: by cutting images out of this sheet and drawing them. I do not recommend this method to anybody. It is ridiculous and requires way too much work. Besides, XNA supports bitmap fonts now, and here's how to use them.

Creating The Project

Assuming you have Visual C# Express installed and XNA Game Studio 1.0 Refresh installed over it, you can go to **File | New Project...** and choose **Windows Game.** Name your project something totally astounding, and click **OK** to create the project.



A project for creating a Windows game using the XNA Framework

Name: TotallyAstoundingPong

Location: C:\Users\dawate\Documents\Visual Studio 2005\Projects

Solution: Create new Solution ☑ Create directory for solution

Solution Name: TotallyAstoundingPong

Important Methods and How This Thing Works

You will notice, after the project is created, that lots of work has already been done for you. In fact, if you press **Ctrl+F5** to run the app without adding any real code, you'll have a game to play already called Cornflower Blue.

So how is our game going to work? Here is a 30,000ft view.

1. We'll initialize our resources by loading assets (images, audio, models, textures, etc.) into XNA's content pipeline by calling the LoadGraphicsContent method.
2. Infinitely, the game will call Update and then Draw in a nicely threaded fashion.
3. When the game is over, it will call UnloadGraphicsContent and quit.

These four methods - LoadGraphicsContent, Update, Draw and UnloadGraphicsContent - should not be removed or have their signatures changed. They all play an integral role in your program. Let's see how.

What do these methods do?

- LoadGraphicsContent - This is where you load assets into the pipeline and have the opportunity to get a variable out of it. For example, in this game I want to know about the texture - the dimensions and such - so I get Texture2D instances as a result of loading sprites. Here's my LoadGraphicsContent method. In my version (in the source code), I also did some initial value initialization that isn't really cohesive with this method, but it works. You can see that the ContentManager class takes care of everything, and I have Texture2D objects that I'm setting as a result of the generic Load method on the ContentManager instance.

The SpriteBatch object is important because it is used to actually draw sprites in the Draw method.

```
protected override void LoadGraphicsContent(bool loadAllContent)
{
      if (loadAllContent)
      {
            spriteBatch = new SpriteBatch(this.graphics.GraphicsDevice);

            // Load the images into a texture object
            ContentManager contentLoader = new
      ContentManager(this.Services);
            texPongoid       =
      contentLoader.Load<Texture2D>("Sprites\\Pongoid") as Texture2D;
            texPaddleLeft    =
      contentLoader.Load<Texture2D>("Sprites\\LeftPaddle") as Texture2D;
            texPaddleRight   =
      contentLoader.Load<Texture2D>("Sprites\\RightPaddle") as Texture2D;
            texNumbers =
      contentLoader.Load<Texture2D>("Sprites\\NumberMap") as Texture2D;
            texHeader =
      contentLoader.Load<Texture2D>("Sprites\\PongHeader") as Texture2D;
      }
}
```

- Update - This method is where it's at. It's where all the cool kids hang out. I normally break out into subroutines here, but I suppose if you have a penchant for really cluttered code, you could put almost all of your game logic in here. Common things to do here include: getting keyboard input, updating sprite positions, updating scores, checking for collision detection, updating AI, etc. The Update method occurs **once every frame**, and consists of the **updating** that is done **before drawing** to generate that frame.

  Here is (the stripped down version of) my Update method. Hopefully all the subs in here are self-explanatory, except for maybe UpdatePongoid (which moves the ball) and Audio.Update (the XNA audio engine needs to be explicitly updated every frame).

```
protected override void Update(GameTime gameTime)
{
      // Allows the game to exit
      if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
ButtonState.Pressed)
            this.Exit();

      CheckKeyboardInput();
      CheckCollisions();
      UpdatePongoid();
      UpdateAIPaddle();

      Audio.Update();

      base.Update(gameTime);
}
```

- Draw - Now that you've updated everything in the game world, you can draw those objects on the screen. **Beware -** there is a lot of math here. When you see foursomes of numbers together in these methods, for example, **0, 0, blah.width, blah.height**, all that means is "Top left coordinates: 0,0. Size of the rectangle is width, height." When you see me dividing things by two, that's because I'm trying to find the center (since the coords start at the top left). Easy, right? Note that the screen is cleared to white and drawn upon (from the bottom up, in the order of the instructions) **every frame**. GDI couldn't hold a candle. Here's some code:

```
protected override void Draw(GameTime gameTime)
{
        graphics.GraphicsDevice.Clear(Color.White);

        // Use alpha blending to utilize the transparency
        spriteBatch.Begin(SpriteBlendMode.AlphaBlend);

        // Background
        spriteBatch.Draw(texBackground, new Rectangle(0, 0, viewportWidth, viewportHeight),
Color.White);

        // Left paddle
        spriteBatch.Draw(texPaddleLeft, new Rectangle(paddleLeftPos, paddleLeftY, 18, 133),
Color.White);

        // Right paddle
        spriteBatch.Draw(texPaddleRight, new Rectangle(paddleRightPos, paddleRightY, 18,
133), Color.White);

        // Header (Pong?!)
        spriteBatch.Draw(texHeader, new Rectangle(0, 0, texHeader.Width, texHeader.Height),
Color.White);

        // Player 1 score
        spriteBatch.Draw(texNumbers, new Rectangle(texHeader.Width, 54, 40, 40),
GetNumberTextureRect(p1score), Color.White);

        // Player 2 score
        spriteBatch.Draw(texNumbers, new Rectangle(texHeader.Width, 94, 40, 40),
GetNumberTextureRect(p2score), Color.White);

        // Pong-oid (gamepiece)
        spriteBatch.Draw(texPongoid, new Rectangle(pongoidX, pongoidY, texPongoid.Width /
2, texPongoid.Height), new Rectangle(textureStartX, 0, texPongoid.Width / 2,
texPongoid.Height), Color.White);

        // End the sprite batch
        spriteBatch.End();

        base.Draw(gameTime);
}
```

- Finally, we clean up. If you just dispose the content manager you'll be fine. If you have any resources that are manually managed, you must dispose of them here. Additonally, if you are the type that insists on disposing every object that's ever been instantiated in the CLR, this is the place to do it.

```
protected override void UnloadGraphicsContent(bool unloadAllContent)
{
        if (unloadAllContent)
        {
                // TODO: Unload any ResourceManagementMode.Automatic content
                content.Unload();
        }

        // TODO: Unload any ResourceManagementMode.Manual content
}
```

Making The Paddle Move

For now, we are making a Player vs. CPU style game, where you move your paddle and the program moves its own. In our case, we set a speed that our paddle moves at, and we handle some keyboard events.

This is what happens in code when we press the up or down keys:

```
protected void CheckKeyboardInput()
{
        //Get the current state of the keyboard
        KeyboardState aKeyboard = Keyboard.GetState();

        //Get the current keys being pressed
        Keys[] aCurrentKeys = aKeyboard.GetPressedKeys();

        //Cycle through all of the keys being pressed and move the sprite accordingly
        for (int aCurrentKey = 0; aCurrentKey < aCurrentKeys.Length; aCurrentKey++)
        {
                //Change the sprite's direction to up and move it up
                if (aCurrentKeys[aCurrentKey] == Keys.Up)
                {
                        if (ySpeedP1 > 0)
                        {
                                ySpeedP1 *= -1;
                        }
                        paddleLeftY += ySpeedP1;
                }

                //Change the sprite's direction to down and move it down
                if (aCurrentKeys[aCurrentKey] == Keys.Down)
                {
                        if (ySpeedP1 < 0)
                        {
                                ySpeedP1 *= -1;
                        }
                        paddleLeftY += ySpeedP1;
                }

                //Exit the game if the Escape key has been pressed
                if (aCurrentKeys[aCurrentKey] == Keys.Escape)
                {
                        this.Exit();
                }
        }
}
```

If ySpeed is negative, the paddle moves up (down if positive). ySpeed is also just a measure of pixels. If ySpeed is 5, the paddle will jump by 5 pixels every frame (which is quite fast). You can also check the time that has passed since the last frame and then increment this value; this allows for more consistent performance. Notice that since paddleLeftY is updated here, so when Draw is called, it will be drawn with the updated position.

Making Pongoid Move

He doesn't want to, judging by the look on his face. But unfortunately, it's his destiny.

Rather than mess with angles and reflection for a truly accurate model of pong (notice there is also no acceleration or gravity here) I took the simple way out. All the UpdatePongoid method does is set his position to the latest calculated position. The real magic happens in the collision detection, which we'll talk about in a moment.

How about right now?

Collision Detection

You can do 2D collision detection in 2 ways. The first way is easy to understand and a pain to implement. The second way is a pain to understand and easier to implement.

Method 1 is pretty standard in games. You check the boundaries of Sprite 1 with the boundaries of all the other sprites to find out if they intersect. This means writing a lot of code that says "if the left side of this is less than the left side of the wall, but not if the top of this is between the Y values of the top and bottom of the paddle." Tedious, yes, but it's easy to understand.

The second way is by using BoundingBoxes, which are used for collision detection in 3D. You can just zero out the Z value and use the nice Intersect method to determine if there is a collision.

I used BoundingBoxes and a little bit of Method 1 when I needed to do something other than bounce the ball. The CheckCollisions method is by far the longest, because it is here that almost all of my game logic is defined. So, here are some important parts of it:

```
protected void CheckCollisions()
{
        // Create bounding boxes for the paddles and pongoid
        BoundingBox bbPongoid = new BoundingBox(new Vector3(pongoidX, pongoidY, 0), new
Vector3(pongoidX + texPongoid.Width / 2, pongoidY + texPongoid.Height, 0));
        BoundingBox bbPaddleLeft = new BoundingBox(new Vector3(paddleLeftPos, paddleLeftY, 0),
new Vector3(paddleLeftPos + texPaddleLeft.Width, paddleLeftY + texPaddleLeft.Height, 0));
        BoundingBox bbPaddleRight = new BoundingBox(new Vector3(paddleRightPos,
paddleRightY, 0), new Vector3(paddleRightPos + texPaddleRight.Width, paddleRightY +
texPaddleRight.Height, 0));
        BoundingBox bbMainBox = new BoundingBox(new Vector3(50, 0, 0), new
Vector3(viewportWidth - 50, viewportHeight, 0));

        // <SNIP>

        // Determine what to do if it is hitting a side
        if (pongoidY <= 0) // the pongoid hit the ceiling, bounce it downward
        {
                Audio.Play("side_hit");

        // set the direction of the pongoid to -1
         // if the speed is currently -2, the pongoid will now move down
         // with a speed of +2 pixels per frame.
                pongoidVSpeedMultiplier = -1;
        }

        // <Lots of Snippage>
    // change direction based on a collision with the left paddle
    if (bbPongoid.Intersects(bbPaddleRight))
    {
            // Play the right boop
            Audio.Play("right_paddle");

            if (pongoidSpeed < 0)
            {
                    pongoidSpeed *= -1;
            }
            if (pongoidCenterY > paddleRightCenterY)
            {
                    pongoidVSpeedMultiplier = -1;
            }
            else
            {
                    pongoidVSpeedMultiplier = 1;
            }
    }

    // <Gratuitous Snippage>
}
```

The code that goes into <Lots of Snippage> and <Gratuitous Snippage> is a little convoluted and I could have certainly written it better. But I think the code here and comments sums up the idea: Check for collision using bounding boxes, and do stuff accordingly. There is a place in the code where, based on where the pongoid hits a paddle (how far from the center it hits), the angle of reflection is impacted. Except it's not quite that complicated. If the pongoid hits a paddle near the very top or bottom of the paddle, the Y speed is assigned a (relatively) high value of 4 or so. If the pongoid contacts the paddle closer to the middle 1/3rd of the pad, the Y speed is assigned an intermediate value of 2. If the pongoid hits near the center, it is assigned a Y speed of 1, which produces a very light angle of reflection.

All of these calculations and modifications take place in the CheckCollisions method. Much of that logic could probably use some cohesion or movement into some subroutines.

Artificial Intelligence (sic)

This is the most basic AI one could possibly produce: "I am going to move toward a certain object." A more detailed description of that process would be "If the ball is higher than the paddle, move the paddle up by a restricted amount." Of course, we have to limit the speed or else it will be impossible to win.

Here's some (laugh) AI for the opponent's paddle:

```
private void UpdateAIPaddle()
{
      int pongoidCenterY, paddleRightCenterY;
      pongoidCenterY = pongoidY + (texPongoid.Height / 2);
      paddleRightCenterY = paddleRightY + (texPaddleRight.Height / 2);

      // Move up or down? Or not at all?
      if (pongoidCenterY > (paddleRightY + texPaddleRight.Height / 2)) // Need
to move paddle down
      {
            paddleRightY += ySpeedP2;
      }
      else if (pongoidCenterY < paddleRightY + texPaddleRight.Height / 2) //
Move paddle up
      {
            paddleRightY -= ySpeedP2;
      }
}
```

This is very simple AI, but it's a start. The more advanced the AI, the more challenging and fun the game is. Because I have limited reflection angles and very predictable AI, you'll find that the only time you can really lose is when you are caught completely off guard.

Conclusion

XNA is an extremely powerful platform for rapid game development. I build a whole game - albeit very simple - from end-to-end in about 10 hours (including art and sound). Moving to 3D will be challenging but entertaining, and the engine is powerful enough to really have some fun!

Download the source: Please leave a comment if you'd like to see the source. If I get ten comments or more on this article I'll post the game :)