

Coding for Game Development

Section Eight

XNA Game Programming

Object Oriented Sprite Handling [Part Two]

1 / 23

A Moving Sprite

- The problem with the basic sprite is that it doesn't actually move!
- We can extend its functionality so that it moves across the screen for every update.

2 / 23

Sprite Velocity

- A 2D sprite can move in *three* directions!
 - X velocity in pixels per update (**double**)
 - Y velocity in pixels per update (**double**)
 - Rotation in radians per update (**double**)
- If we define these three fields, then we can modify the **Update()** method (within our class) so that the sprite automatically moves without any further intervention.

3 / 23

The MovingSprite Class

- Our first try at a **MovingSprite** class looks like this:

extends **BasicSprite**

```
class MovingSprite : BasicSprite
{
    double _velocityX=0.0, _velocityY=0.0;
    double _rotation = 0.0;
    : : :
}
```

- This class automatically has all the fields and methods of **BasicSprite** – plus three velocities.

4 / 23

The Update() Method

- The **Update()** method moves the sprite according to the various velocities. It does this by accessing XY position properties defined in **BasicSprite**.

```
public override void Update() // For MovingSprite class
{
    // Move the positions
    this.X += _velocityX;      // x position property of BasicSprite
    this.Y += _velocityY;      // y position property
    this.Angle += _rotation;   // Perform sprite animation
    base.Update();
}
```

- Note: final line is a call to **base.Update()**, which calls the **Update()** in **BasicSprite** and therefore ensures that the fields within **BasicSprite** are also updated.

5 / 23

Edge of the Screen

- The sprite can slide off the edge of the screen without any warning.
- There are several ways we can handle this:
 - Do nothing.
 - We might add bouncing behaviour to **Update()**, so if the sprite hits the edge of the screen then it bounces back.
 - When bouncing behaviour is not appropriate then we might clear the **BasicSprite show** flag when the sprite is invisible. Within the main **Update()** method we can test this flag and take the necessary steps to delete the sprite object.

6 / 23

Setting the Velocities

- We can create three properties to set up the X, Y and rotational velocities. For example:

```
public int VelocityX
{
    get { return _velocityX; }
    set { _velocityX = value; }
}
```

- Once we've created a **MovingSprite** then we can give it a velocity using the properties:

```
MovingSprite sprite = new MovingSprite(...);
sprite.X = 100;
sprite.Y = 200;
sprite.VelocityX = 3;
sprite.VelocityY = 4;
sprite.Rotation = 0.01;
```

7/23

The **MovingSprite** Constructor

- The constructor for **MovingSprite** needs to call the base constructor for **BasicSprite**.
- It should also set the new fields to known values:

```
public MovingSprite(Texture2D image,
    Game game,
    int frameWidth,
    int cycleStart, int cycleEnd, int frameRate,
    int originX, int originY)
: base(image, game, frameWidth,
    cycleStart, cycleEnd, frameRate, originX, originY)
{
    _velocityX = 0.0;
    _velocityY = 0.0;
    _rotation = 0.0;
}
```

call to **BasicSprite** constructor

8/23

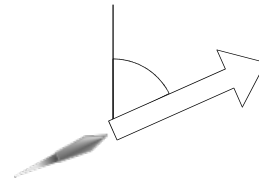
The **Draw()** Method for **MovingSprite**

- The **Draw()** method defined in **BasicSprite** is usually adequate, and we don't need to define a new version for **MovingSprite**.
- Under the rules of object orientation, if we don't define a new method then we automatically get the version defined in the base (parent) class.

9/23

Vectored Sprite

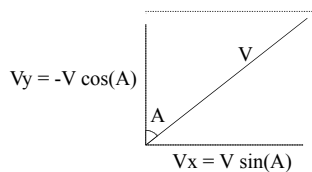
- In Physics, a *vector* is a quantity which has a size *and* a direction.
- A *Vectored Sprite* moves along a particular vector:
 - Angle of movement.
 - Speed or velocity.



10/23

X and Y Velocities

- The vector velocity is in the direction of movement.
- To apply this to a moving sprite, we need to break it down into separate X and Y velocities:



- The minus sign in the V_y formula is required because we number screen pixels from the *top* downwards.

11/23

The **VectoredSprite** Class

- A **VectoredSprite** is a particular form of **MovingSprite**, which moves at some velocity in some direction.
 - The direction of motion is given by the rotation angle previously defined in **BasicSprite**.
 - The velocity is a new field defined within the class.
- The X and Y velocity are obtained by using sine and cosine:
 - $xVel = velocity * \text{Math.cos}(angle)$;
 - $yVel = -velocity * \text{Math.sin}(angle)$;
- The cos and sin functions produce **double** results which works well with **double** X/Y velocities and positions.

Radians!

12/23

The **VectoredSprite** Constructor

- The constructor for **VectoredSprite** is similar to that for **MovingSprite**:

```
public VectoredSprite(Texture2D image,
                    Game game,
                    int frameWidth,
                    int cycleStart, cycleEnd, frameRate,
                    int originX, int originY)
: base(image, game, frameWidth,
      cycleStart, cycleEnd, frameRate, originX, originY)
{
    _velocity = 0.0;
}
```

Calls **MovingSprite**
constructor

Calls **BasicSprite**
constructor

13 / 23

The **SetVector()** Method

- The sprite's vector is defined by both its velocity and its direction (angle).
- We will automatically inherit the **Angle** property from the base **BasicVector** class, and we should also add a further property for the **Velocity**.
- It is also convenient to define a method update both of these components in one call:

```
public void SetVector(double velocity, double angle)
{
    _velocity = velocity;
    Angle = angle;
}
```

Angle property defined in **BasicSprite**

14 / 23

The **Update()** Method (of **VectoredSprite**)

- To ensure that the sprite always moves in the correct direction, the **VectoredSprite Update()** method must recalculate the X and Y velocities every time.
- It must then call **base.Update()** to actually move the sprite position and perform any animation steps.

```
public override void Update()
{
    // Recalculate the X and Y velocities
    this.VelocityX = _velocity * Math.Sin(this.Angle);
    this.VelocityY = -_velocity * Math.Cos(this.Angle);

    // Move the sprite and animate it
    base.Update();
}
```

15 / 23

The **Draw()** Method

- As with **MovingSprite**, we probably don't need to define a new version of **Draw()** for **VectoredSprite**.

16 / 23

Putting it All Together

- The various sprite objects still have to be created within the XNA boilerplate.
- The complex set of methods within the boilerplate can make it difficult for the novice to know where to do certain actions. However, this complexity can be tamed by using some simple rules:
 - Declare all sprite objects, 2D Textures, state and score variables as fields of the **Game** class.
 - Load all textures within the **LoadContent()** method.
 - Create **new** sprite objects inside **Update()** and nowhere else!
 - Never use the **Initialize()** method for 2D games!!

17 / 23

Creating and Destroying Sprite Objects

- During the course of the game, sprite objects will appear and disappear.
- Note: all sprite changes should be triggered by our Finite State design.
- We can make a sprite *appear* by using this code in the boilerplate **Update()** method:

```
spriteObject = new XxxxxxSprite(....);
spriteObject.X = ...;
spriteObject.Y = ...;
```

- We can make a sprite *disappear* by using this code in the boilerplate **Update()** method:

```
spriteObject = null;
```

18 / 23

The Game Update() Method

- Within the game **Update()** method, we need to make calls to **Update()** methods of each sprite.
- We probably need to guard each call with an **if** statement to prevent a null pointer exception.
- We'll also need to add extra code for collision detection, edge of screen detection, end of game, etc.

```
if (theSpriteOne != null) theSpriteOne.Update();  
if (theSpriteTwo != null) theSpriteTwo.Update();  
: : :
```

19 / 23

The Boilerplate Draw() Method

- Within the boilerplate **Draw()** method, we usually call the **Draw()** methods for each individual sprite.
- We must guard each call with an **if** statement to prevent a null pointer exception.

```
protected override void Draw(GameTime gameTime)  
{  
    graphics.GraphicsDevice.Clear(Color.CornflowerBlue);  
  
    // TODO: Add your drawing code here  
    spriteBatch.Begin();  
  
    if (theFirstSprite != null) theFirstSprite.Draw(spriteBatch);  
    if (theSecondSprite != null) theSecondSprite.Draw(spriteBatch);  
    if (theThirdSprite != null) theThirdSprite.Draw(spriteBatch);  
  
    spriteBatch.End();  
    base.Draw(gameTime);  
}
```

20 / 23

Why not Initialize() ?

- It's not immediately obvious why we should create new sprite objects within the boilerplate **Update()** method as opposed to **Initialize()**.
- However, many items need to be reset and re-initialised when a new game starts, or when the game changes to a different level.
- The XNA framework is ignorant of these changes in state and hence won't call **Initialize()** when we might need it.
- The only method which is reliably called in each case is **Update()**

21 / 23

Gotchas

- Many students are able (with a bit of effort) to create a game which starts ok and finally reaches a conclusion, but they then struggle to start a new game without restarting the application:
- Common difficulties include:
 - Inability to handle separate "INSERT COIN", game play, game levels, and "GAME OVER" screens.
 - Forgetting to reset score to zero at the start of a new game.
 - Forgetting to reset to level one at the start of a new game.
- Most of these issues can be handled by using an appropriate Finite State design.

22 / 23

23 / 23