

## Coding for Game Development

### Section Seven

#### XNA Game Programming

#### Object Oriented Sprite Handling [Part One]

1 / 21

## Object Oriented Sprites

- Most code examples on the internet use fields of the **Game** class to store a sprite's parameters.
- This gets unwieldy and awkward as the number of sprites increases, and we have to think of unique names for each one.
- A better approach is to encapsulate all the data and behaviour of a sprite into a class, and then create one object for each new sprite.
- This can be readily combined with a Finite State model, simplifying the whole game design process.

2 / 21

## Basic Sprite

- A *basic* sprite has the following obvious fields:
  - A 2D Texture for the image.
  - An X and Y position (two **double** values).
- And some not-so-obvious fields:
  - An origin (ie. the centre point for the image)
  - A rotation angle (in radians?)
  - A colour tint
  - A scaling factor
  - A finished flag (described later)
- For the moment we'll assume that the sprite doesn't move across the screen.

3 / 21

## Sprite Object Behaviour

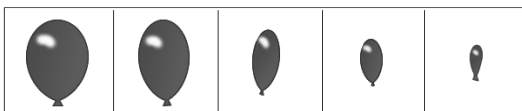
- The XNA boilerplate code uses the methods **Update()** and **Draw()** for updating and drawing a sprite. This separation of behaviour is an important part of XNA.
- For this reason, we must put *two* methods in our Basic Sprite class:
  - **void Update()** called from the boilerplate Update() method.
  - **void Draw(SpriteBatch)** called from the boilerplate Draw() method.
- Within each method, we must put appropriate code to update and draw the sprite.

4 / 21

## Changing Frames

- The texture (ie. graphical image file) of a typical sprite sheet contains multiple images, often called *frames*.
- Changing the frame changes the sprite image.
- We can produce a crude form of animation by cycling through the images, changing the frame every few updates.

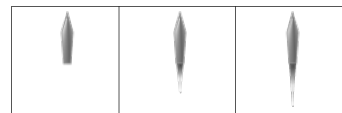
To make our code simpler, we assume that all frames are the same size and occur side-by-side on the sprite sheet.



5 / 21

## Animation Cycles

- Look at this simple sprite sheet, showing three separate sprite images of a missile:



- Frame 0 shows the missile waiting to be launched. In this situation we should show just frame 0.
- Frames 1 and 2 show the missile in flight. In this situation, we should rapidly cycle between frames 1 and 2 to give the effect of a flickering rocket flame.

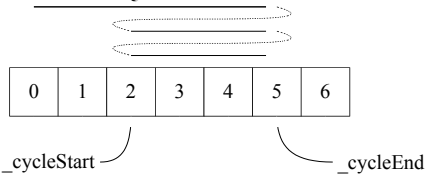
6 / 21

## Cycle Start, End and Frame Rate

- To support an animation cycle, we need four fields:

```
int _currentFrame, _frameRate;
int _cycleStart, _cycleEnd;
```

- Animation starts from the current frame, and changes to the next frame at the specified frame rate.
- When the cycle end frame has been shown, the cycle reverts back to the cycle start frame.



7/21

## Cycle Variations

- Animations which loop in a fixed cycle are useful, but not always appropriate.
- Sometimes we want the animation to stop at a particular frame and for the sprite to disappear:
  - We can achieve this by using -1 as a sentinel value for the cycle start. When the cycle reaches the end frame, if the start frame is negative then a *show* flag is cleared.
  - This is useful for explosion animations where the sprite disappears when finished.
- Sometimes we want the last frame to remain visible:
  - We can achieve this by setting the start and end frames to the same value.

8/21

## Basic Sprite Class Extra Fields [1]

- To support frames and cycling, we can identify the following fields to be added to the **BasicSprite** class:
  - Frame Width and Height in pixels: this is the size of each separate frame on the sprite sheet.
  - Frame Number: the current frame being displayed, counting starts from zero.
  - Frame Count: total number of frames on the sprite sheet.
  - Frame Rate: how often to update to the next frame, measured in 1/60<sup>th</sup> second counts.
  - Cycle Start and End frames.

9/21

## Basic Sprite Class Extra Fields [2]

- It is convenient to store the following additional fields in the **BasicSprite** class:
  - A reference to the XNA Game class. This gives access to some useful values, including the window width and height. It is convenient if we mark this particular field as **protected** to allow child classes to access it directly.
  - A **bool** show/hide flag which can be used to stop the sprite being shown on the screen. This flag will be set to hide when the animation finishes and the start cycle value is negative; also it can be used to temporarily hide the sprite for other reasons.

10/21

## The BasicSprite Class

```
class BasicSprite
{
    Texture2D _image;
    protected Game _game;
    Vector2 _origin;
    int _frameWidth, _frameHeight;
    int _frameNumber;
    int _frameCount;
    int _cycleStart, _cycleEnd, _frameRate;
    int _updateCount;
    double _positionX, _positionY;
    bool _finished;
    bool _show;

    Color _tint;
    double _angle;
    double _scale;

    : : : : :
}
```

11/21

## Properties

- We should create a *property* for each of these fields:

```
public double X
{
    get { return _positionX; }
    set { _positionX = value; }
}
public double Y
{
    get { return _positionY; }
    set { _positionY = value; }
}
public double Angle
{
    get { return _angle; }
    set { _angle = value; }
}
: : : etc : : :
```

12/21

## The BasicSprite Update() method

- The role of the **Update()** method is to change to the next image frame when necessary.

```
// Update and cycle through the next frame
public virtual void Update()
{
    // Count the number of updates
    _updateCount++;

    // Is it time to move on to the next frame?
    if (_updateCount > _frameRate)
    {
        _updateCount = 0;
        IncrementFrame(); // Described later
    }
}
```

- Remember this sprite doesn't move across the screen!

13/21

## The IncrementFrame() Method

- Our sprite **Update()** method calls **IncrementFrame()** to control the animation:

```
// method to increment the frame number
private void IncrementFrame()
{
    _frameNumber++;
    if (_frameNumber > _cycleEnd)
    {
        if (_cycleStart >= 0)
        {
            _frameNumber = _cycleStart;
        }
        else
        {
            _show = false;
        }
    }
}
```

14/21

## The Finished Property

- In an earlier slide, we identified a flag named **\_finished** which we should associate with a **Finished** property:

```
bool _finished;
public bool Finished
{
    get { return _finished; } protected set { _finished = value; }
}
```

- This becomes useful when we integrate object oriented sprites with finite state machines.
- When the sprite enters the END state, it should set the **Finished** property to **true**.
- The **Game1** class can check this flag on each update and remove the sprite when it has nothing more to do.

15/21

## The Draw() Method

- The **Draw()** method displays the sprite at its current location, using the supplied **SpriteBatch**

```
// Draw the sprite
public virtual void Draw(SpriteBatch spriteBatch)
{
    // Only draw the sprite if the show option is on.
    if (_show)
    {
        // Set up the call and draw the sprite
        Vector2 position =
            new Vector2((float) _positionX, (float) _positionY);
        Rectangle sourceRect =
            new Rectangle(_frameNumber * _frameWidth, 0,
                _frameWidth, _frameHeight);

        spriteBatch.Draw(_image, position, sourceRect, _tint,
            (float) _angle, _origin, (float) _scale,
            SpriteEffects.None, 0.5f);
    }
}
```

16/21

## The Constructor Method

- We need a constructor method, which has the following heading:

```
public BasicSprite(Texture2D image,
    Game game,
    int frameWidth,
    int cycleStart, int cycleEnd, int frameRate,
    int originX, int originY)
```

- The constructor must calculate the number of frames from the 2D texture, thus:  
    \_frameCount = image.Width / frameWidth;
- The constructor must also initialise the rest of the fields to known values.

17/21

## Creating a BasicSprite Object

- We expect that all sprite objects will be created inside the **Update()** method of the game class.
- When calling the constructor it is necessary to provide a value for every parameter.
- A typical call will look like this:

```
sprite = new BasicSprite(imageTexture, // Sprite sheet image
    this, // Game class reference
    frameWidth, // From sprite sheet
    0, 5, 10, // Animation cycle
    48, 48); // Frame centre
```

Cycle between frames 0 to 5,  
changing every 10 updates

18/21

## Changing the Animation Cycle

- It is useful to be able to change the animation cycle after the sprite has started, so we add a further method:

```
public void setCycle(int cycleStart, int cycleEnd, int frameRate)
{
    _updateCount = 0;
    _cycleStart = cycleStart;
    _cycleEnd = cycleEnd;
    _frameRate = frameRate;

    // Check that we don't run off the end of the sprite sheet
    if (_cycleEnd > _frameCount)
    {
        _cycleEnd = _frameCount;
    }
}
```

19/21

## BasicSprite Collision Detection

- A useful extension to **BasicSprite** is collision detection.
- Here is one possible way of implementing this:
  - Automatically calculate a sprite bounding box in the constructor, and after every change in rotation.
  - Define a method with this heading:  
**public virtual bool Collide(BasicSprite other)**
    - This method returns **true** if **this** sprite is colliding with the **other** sprite, which is passed as a parameter.
- Within the XNA boilerplate **Update()** method, we can test each sprite against each other to find those which have collided.

20/21

21/21