Platform Games
Drawing Sprites & Detecting Collisions

Computer Games Development

CSCU9N6

Introduction

Contents

- Drawing Sprites
- Collision Detection
- Animation Loop

Background Image - Parallax Scrolling

A static background image is a bit dull, speeding up and slowing down the background image and tiles produces a good sense of motion.

- If we move the background image slower than the foreground tiles, we get parallax scrolling.
- The background seems further away because it moves slower
- Racing games use this trick a lot

We will not want to create a background image the size of our tile map.

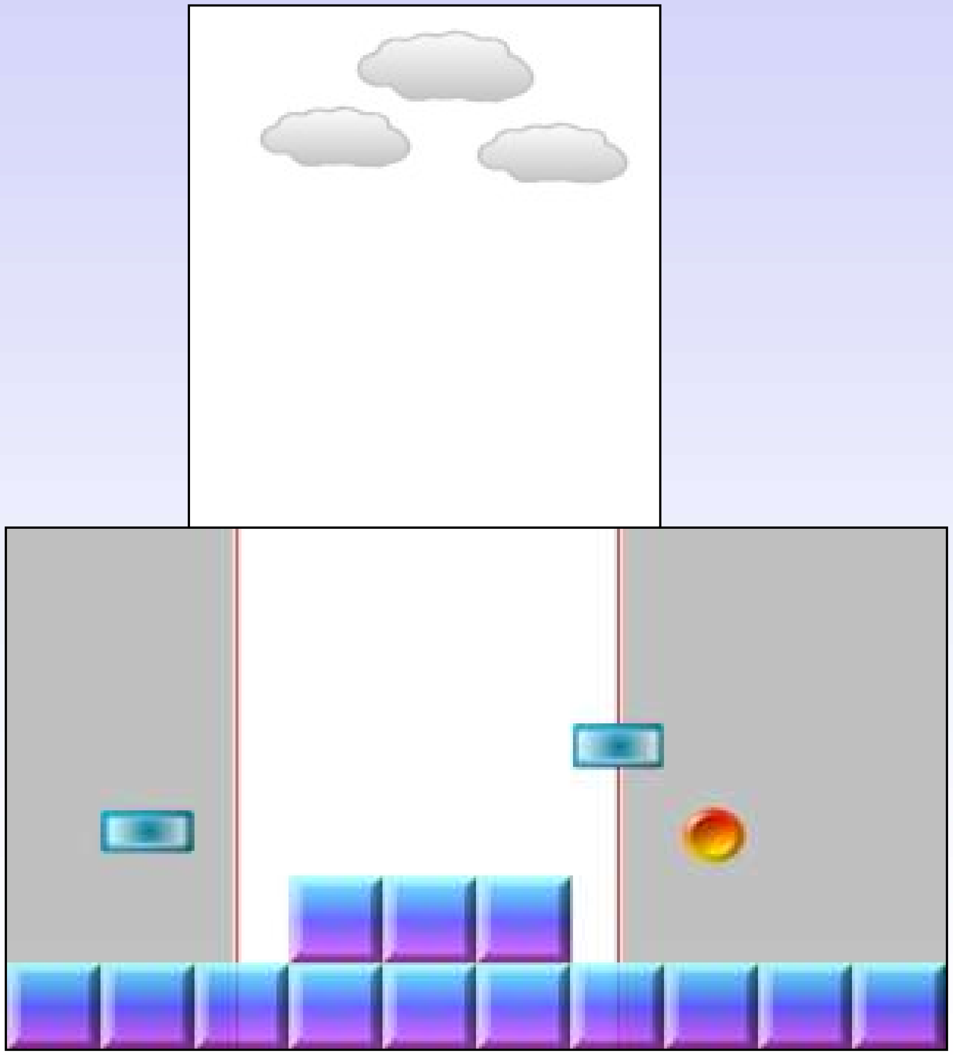
- Instead, create an isomorphic background that is bigger than the viewable area
- Display the visible section as the background
- When the image is about to reach its edge, display the beginning again
- Alternatively move the background so slowly such that the leftmost side is seen at the start of a map and the rightmost side at the end of a map (DGJ, p234-p236)
- Multiple backgrounds, each scrolling at different speeds...

Background Image

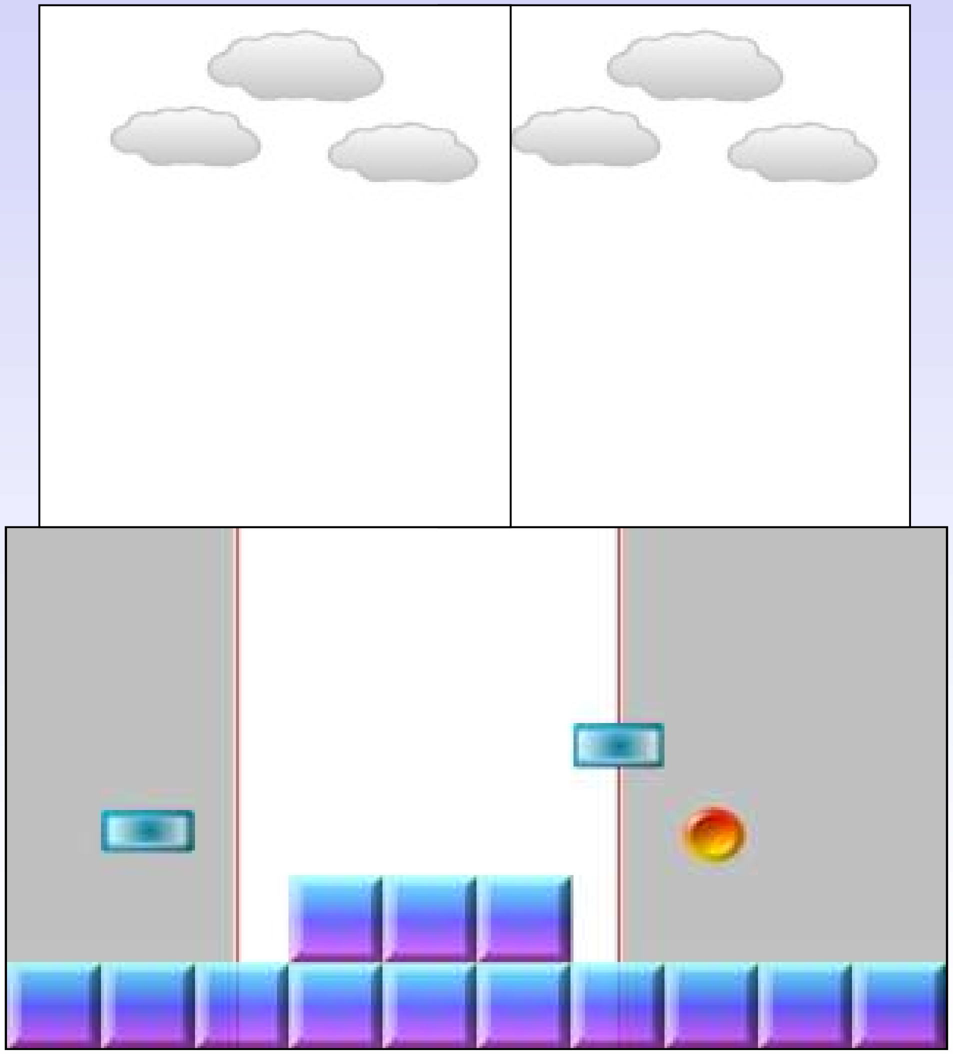
Create a background image larger than the viewable screen

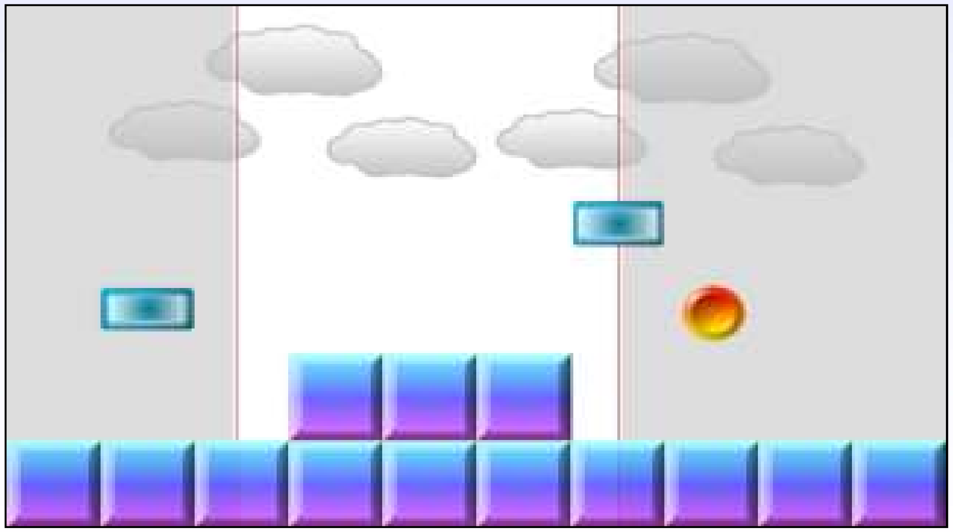
Display the relevant section

Use the image as though it were on a roll, repeating itself at an edge



Repeating Background Image





Drawing Sprites

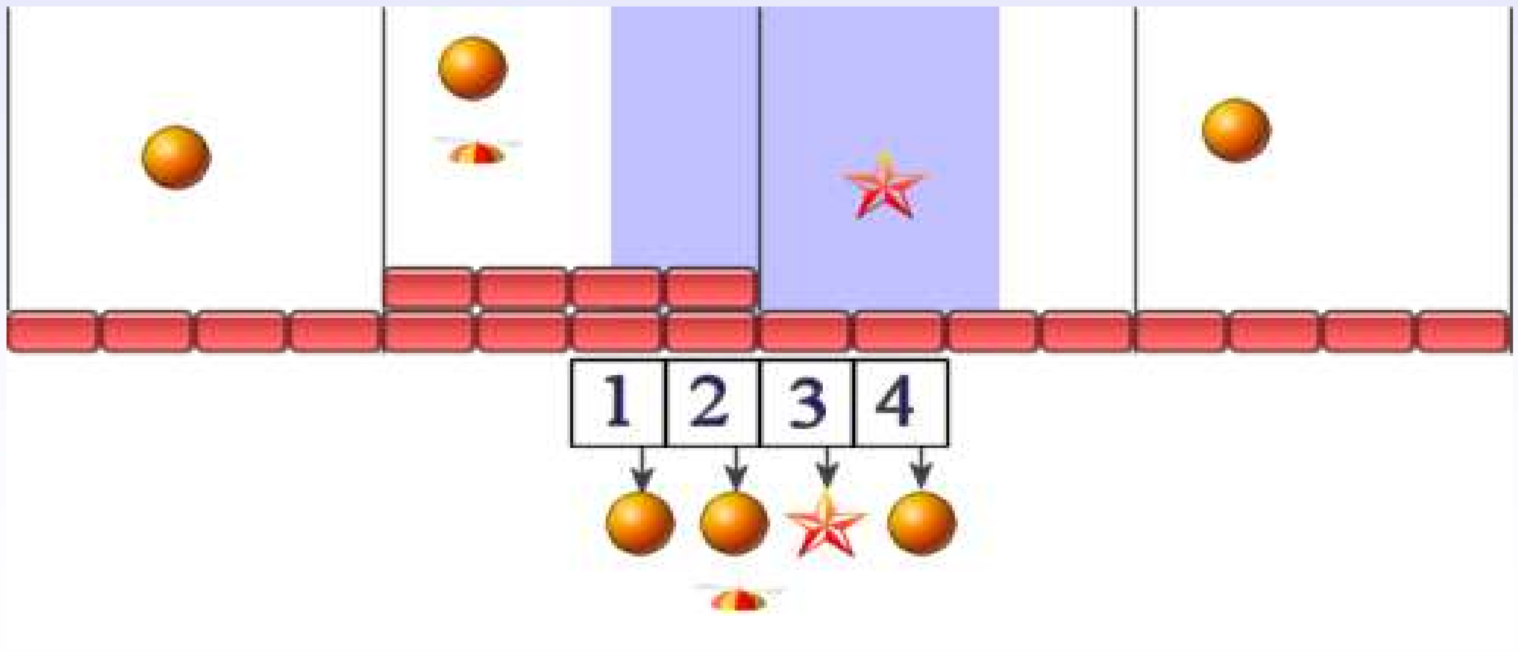
Drawing Sprites

- We already know <u>how</u> to draw animated sprites
- The problem is knowing <u>when</u> to draw them
- Solutions
  - Section up the play area
  - Ordered horizontal list
  - Unordered list

Drawing Sprites - Sectioning

Sectioning

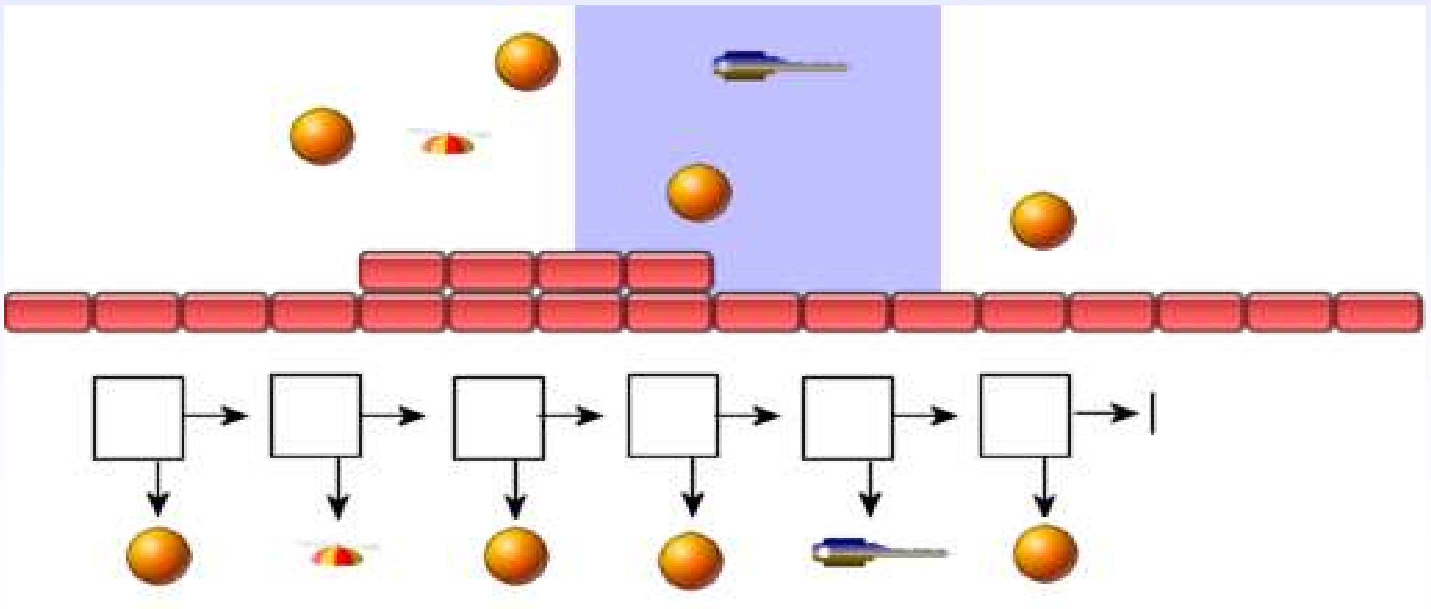
- Section up the game area
- Calculate which sections are currently visible
- For each visible section, draw the sprites it contains
- With this approach we must note when a sprite moves into a new section



Drawing Sprites - Ordered List

Ordered List

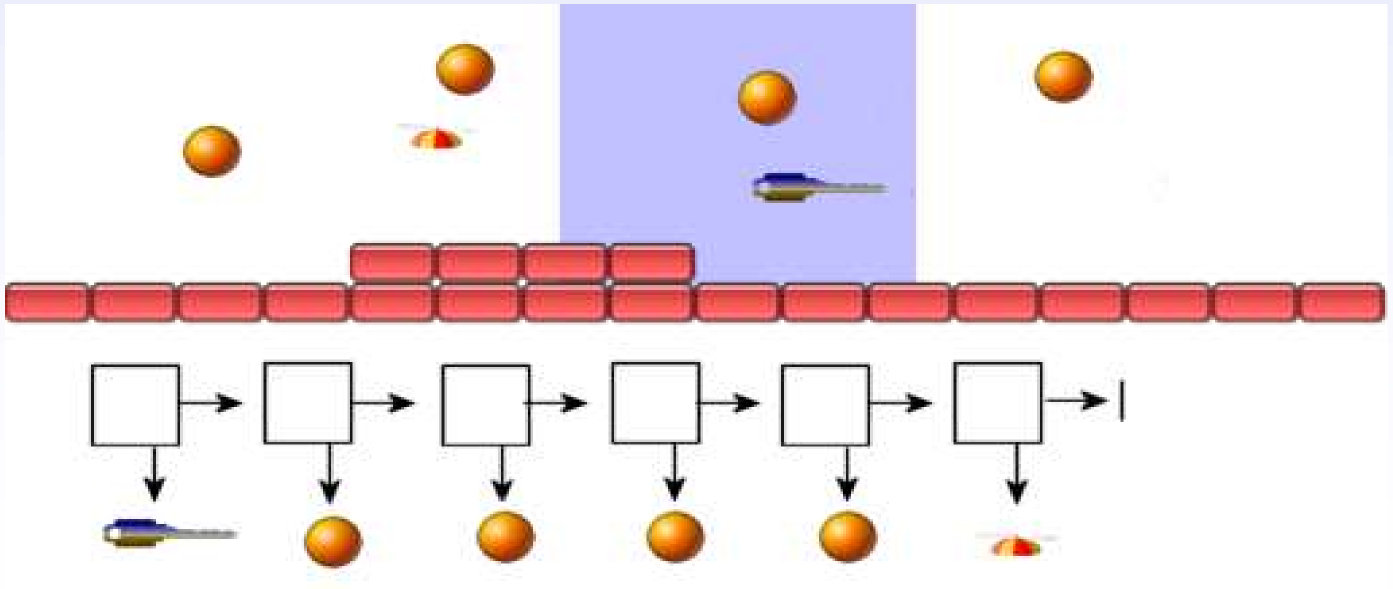
- Create an ordered list of sprites where the leftmost sprite is first and the rightmost sprite is last
- Scan list from the left looking for first visible sprite
- Draw sprites from this one until the rightmost non visible sprite is reached
- Requires constant movement of sprite references up and down the list



Drawing Sprites - Unordered List

Unordered List

- Maintain an unordered list of sprites
- Scan through the entire list every frame and work out if the sprite should be displayed
- Very simple, brute force approach
- Very inefficient for games with a large number of sprites but fine for only 10-20 sprites



Collisions

A game which does not interact with the player would be pretty dull

- If a player jumps on a platform, it would be good for them to stay on it rather than fall straight through
- An interaction between a player sprite and the background (or a monster) is detected as a collision between the objects

Stages

- Determine which objects to test for collisions
- Detect the collisions
- Handle the collisions

Reducing Collision Tests

In a small game with only a few sprites and a tile map, you can easily just check all items for collisions.

In a larger game with perhaps 100 moving objects, you would have to make 4950 comparisons per frame

- $(n * (n-1))/2 = (n^2-n)/2$
- $(100 \times 99)/2 = 4950$

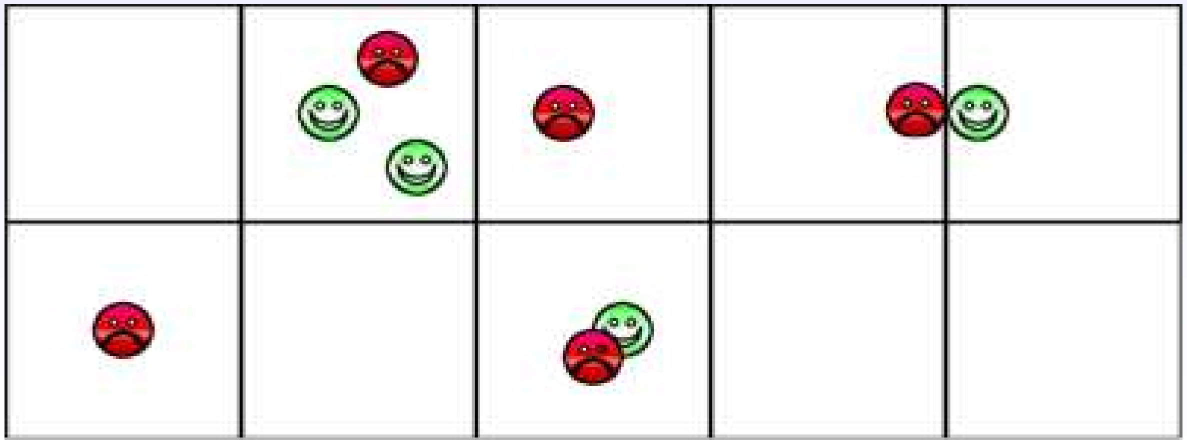
Useful Heuristics

- Do not test static objects, only moving objects
  - Test moving objects that collide with static objects
  - If an object hasn't moved since the last test, don't recheck
- Reduce tests to only those objects within a given area
  - Divide game area into a grid
  - Use an ordered list where items in proximity are close together in the list

Reducing Collision Tests - Grids

Dividing Game Area into a Grid

- Keep track of which objects are in a given grid cell
- Test collisions for objects in current and surrounding cells
- The finer the grid resolution, the fewer objects that will be in each cell therefore the fewer tests you need to make
- Fine resolution requires larger memory requirement
- There is a general overhead in tracking objects in cells which becomes more beneficial the more objects you have.



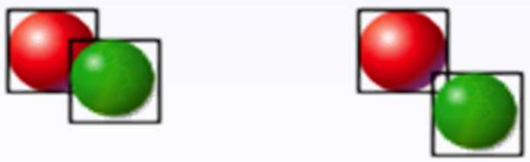
Collision Detection – Bounding Box

Once we have worked out which objects to test for collisions, we must do the collision test

- This is not as simple as you might think / hope

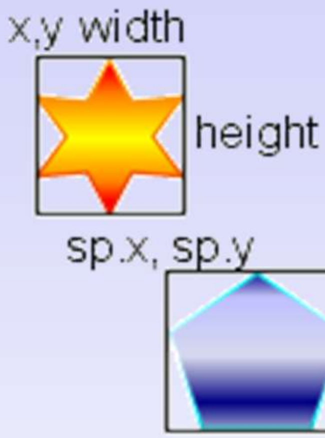
A simple test is to check the bounding boxes of two objects, however this only helps to determine if there <u>might</u> be a collision

- Collision detection usually involves making a number of increasingly detailed checks



Collision Detection – Bounding Box Example

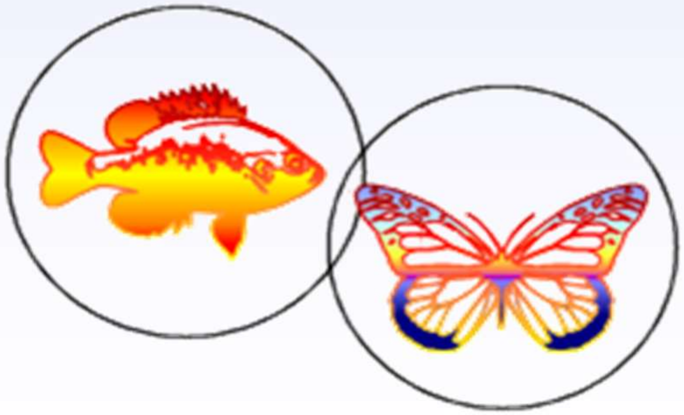
```
public boolean BoundBoxCollision(Sprite sp)
{
        // x,y are current position of this sprite
        // width,height are size of this sprite
        // We now compare this sprite’s values with those passed
        // in via the sprite referenced by ‘sp’
        return ((x + width) => sp.x) && (x <= sp.x + sp.width) &&
                (y + height) => sp.y) && (y <= sp.y + sp.height));
}
```

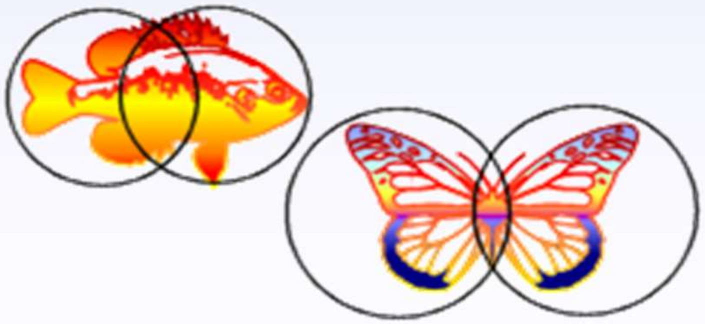


Collision Detection – Bounding Circle

A simple but slightly more sophisticated approach is to use a bounding circle

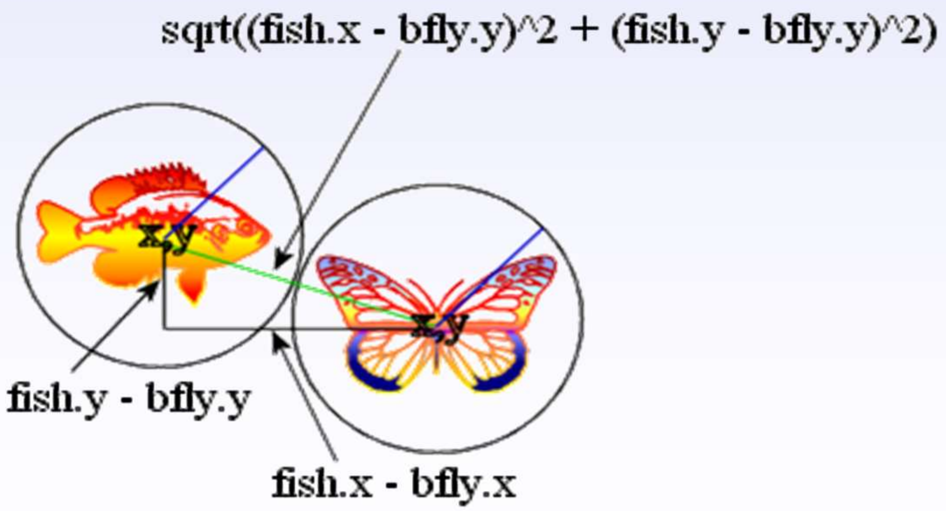
- If an initial collision is suspected, then we can test again on smaller circles, and then even smaller circles
  - In 3D graphics this is commonly called a sphere tree
- Sometimes initial circles are good enough
  - It depends if the user will notice (or get upset) if two objects do not quite visually collide
- The last resort is to test whether the visible pixels in each image overlap – this is very time consuming and generally avoided





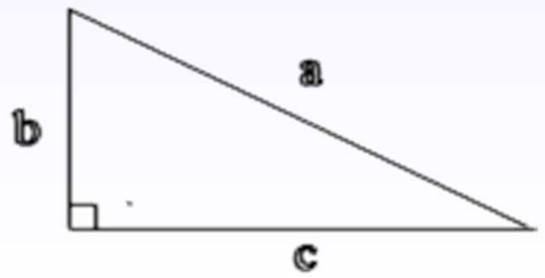
Collision Detection Bounding Circle Example

```
public boolean BoundingCircleCollision(Sprite one, Sprite two) {
            int dx,dy,minimum;

            dx = one.x - two.x;
            dy = one.y - two.y;
            minimum = one.radius + two.radius;

            return (((dx * dx) + (dy * dy)) < (minimum * minimum));
}
```



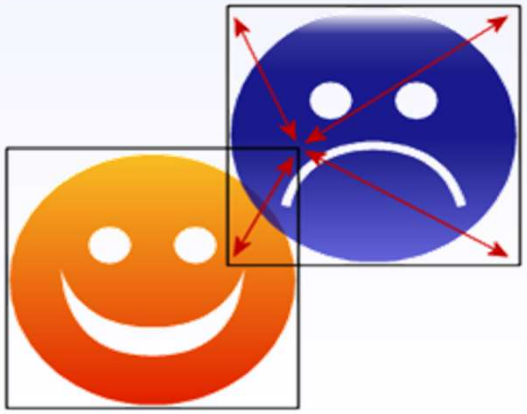
$$a^2 = b^2 + c^2$$

$$a = \sqrt{b^2 + c^2}$$



Detecting collision direction

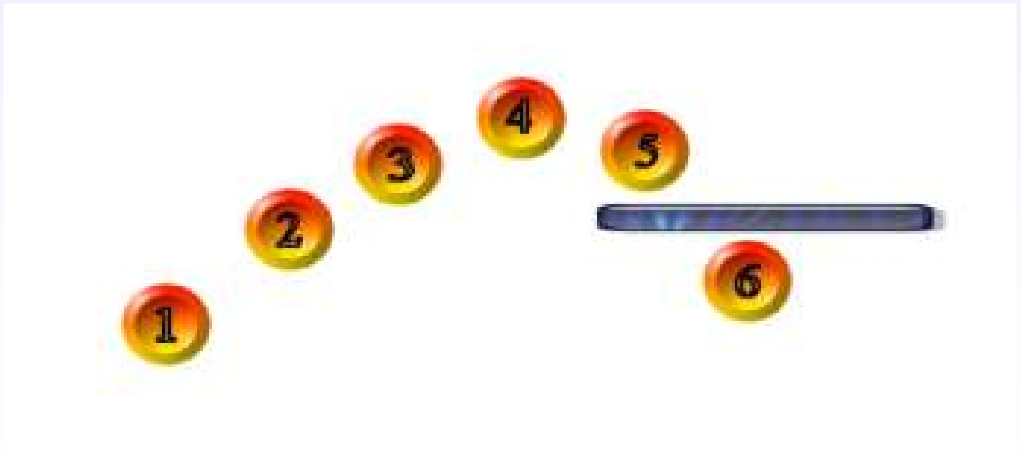
- Determining that there has been a collision is relatively straightforward. Working out which point has collided is a lot harder
  - This is needed if you wish to 'bounce' away correctly
  - A simplified method is to check a number of points around the object and find out which is closest to the object that the collision occurred with (e.g. measure from each corner of the sprite).
  - We can use Pythagoras theorem to measure distance between points, then select the shortest
  - We may also wish to do this for testing collisions of sprites with the corners of tiles on our tile map



Collision Detection & Discrete Time

In most games you will update the sprite positions at discrete points rather than move each sprite a pixel at a time. This can cause a problem!

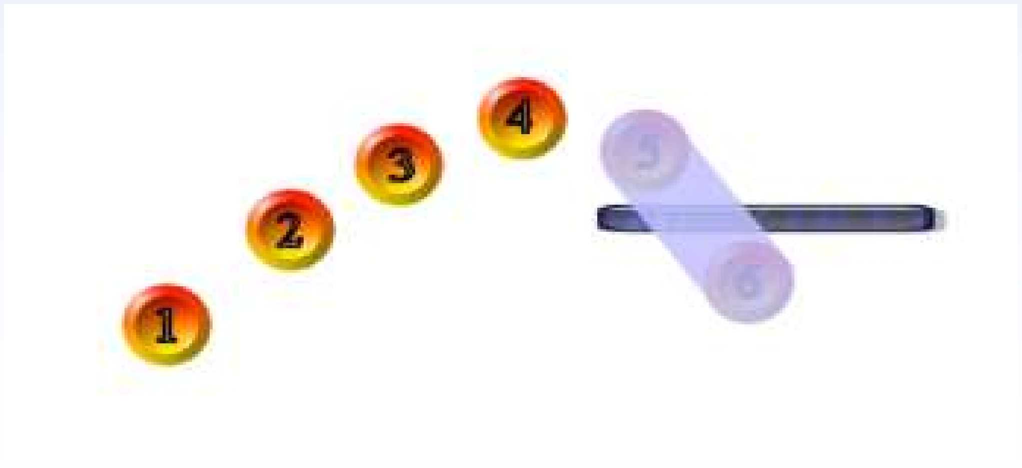
- This is an even bigger issue in multiplayer games where different machines run at different speeds



Collision Detection & Discrete Time

Solutions:

- Compute a bounding shape using the before and after object positions to form the edges
  - *Accurate but computationally expensive*
  - This shape could be simplified to a rectangle (tilted)
- Compute more frames than are actually displayed
  - If we had twice the number of animation frames, a collision would be detected in the new frame between frames 5&6 in the old method



Collision Handling

Once we have detected a collision, we must decide how to deal with it. This is usually done before the colliding objects are drawn. There are two common collision types to handle:

- Collisions with Sprites (e.g. power ups / baddies)
- Collisions with 'physical' objects – e.g. border tiles

Collisions with Sprites

- In this scenario you will usually remove one of the two colliding sprites (the player or a power up), possibly play an animation and a sound.
- For example, with the player colliding with a baddie, you might change the player sprite animation to one where the player 'shrivels up' and play a bad sound to go with it.

Collisions with Objects

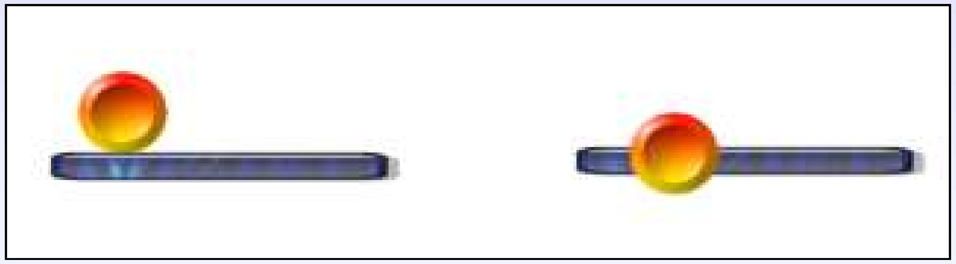
When a sprite collides with a background object, we may wish to:

- Change the position and direction of the sprite depending on the angle of incidence
  - Move the sprite back to a pre-collision position
  - Reverse horizontal and / or vertical velocity
- Alter or remove the tile / object it collided with

Changing velocity (flip)

Collisions - Changing Position

Because we step forward in discrete time intervals (rather than a pixel at a time), we will usually detect a collision when two objects have already overlapped, rather than when they have just touched each other:



Our goal is to work out where the collision occurred and place the sprite back at this point...

Collisions - Changing Position

Having detected the collision, we now move the sprite back to where it started before it moved and then move it a pixel at a time horizontally then vertically (in the direction of its current velocity), until a collision is detected.

The Animation Loop

The animation loop is where we put all these different components together...

- Get user input
- Update sprite positions in game world
- Detect & handle collisions
- Update display
  - Display background image
  - Display tile map
  - Display sprites
- Repeat