CS 134

Actor motion
Pixel perfect collision detection

Today in Video Games

Angry Birds maker Rovio closes London studio

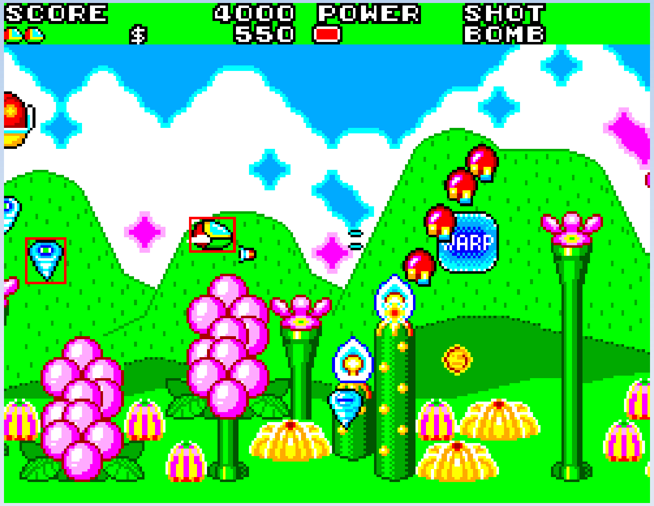
2 March 2018

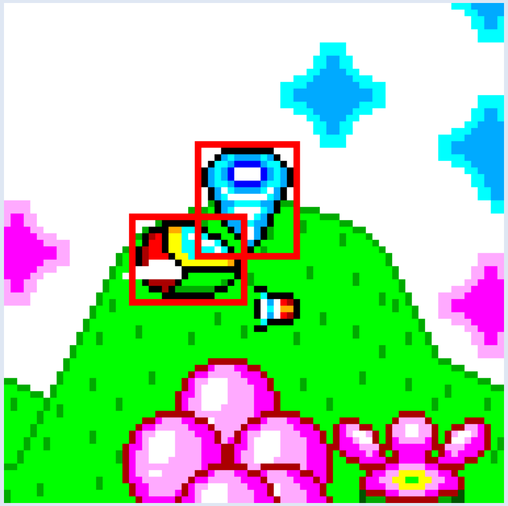
Share

The Angry Birds movie was a commercial success

Pixel Perfect Collision

- For some objects, it is very hard to create a collision shape.
- A “good” fit will still have lots of holes.
  - Lots of false positives
  - Lots of false negatives
- Better if we could test individual pixels



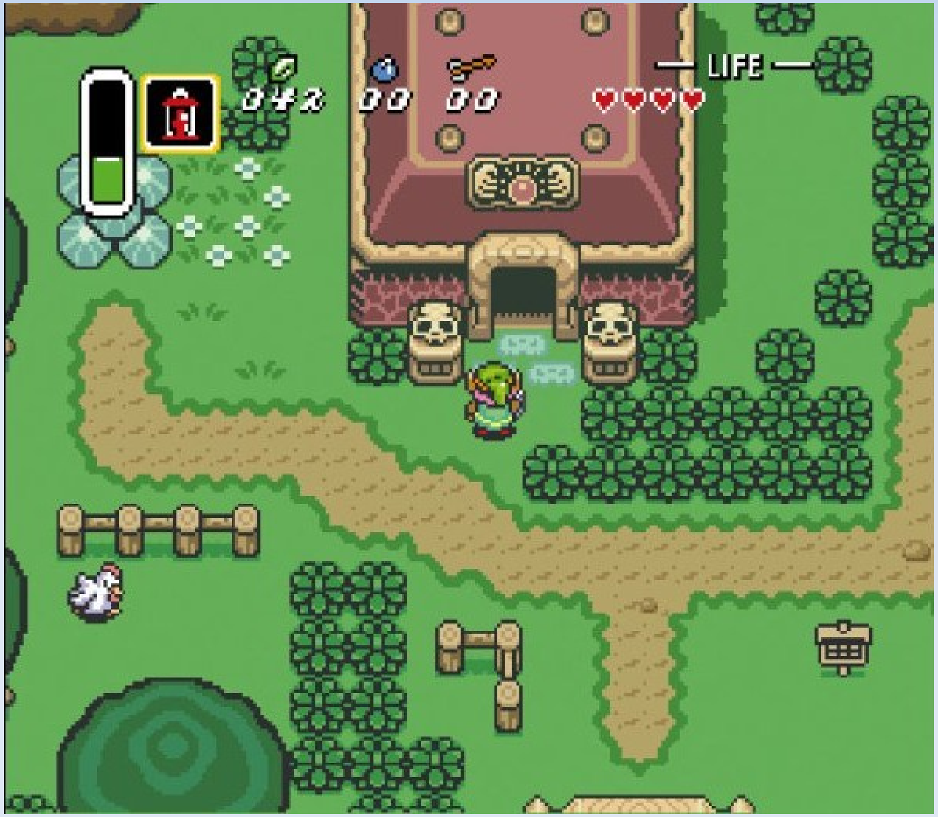


Actor Motion

- Basic motion from last class isn't enough for many games
  - In maze games, the player should be given some give to going around corners
  - In platformers, the player should be affected by gravity
  - One way walls are a common thing too

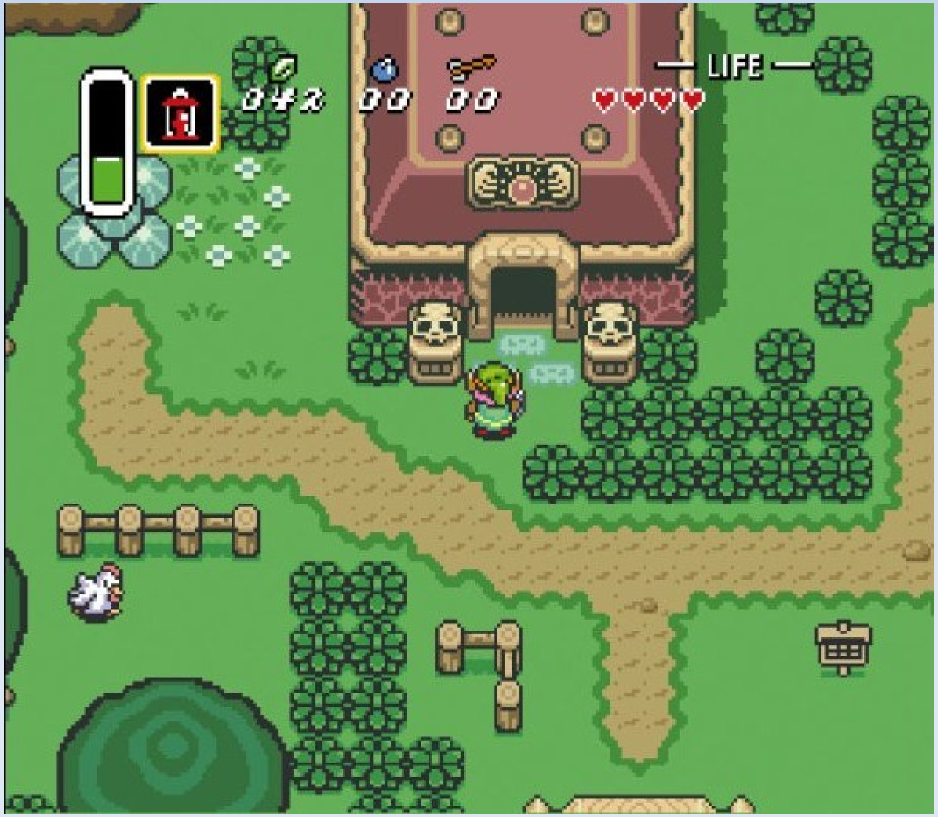
Actor Motion

- Maze edge motion:
  - When moving, if you are just barely colliding, fix the motion to a tile boundary
- Here, pressing up will still get Link into the fortune teller's house



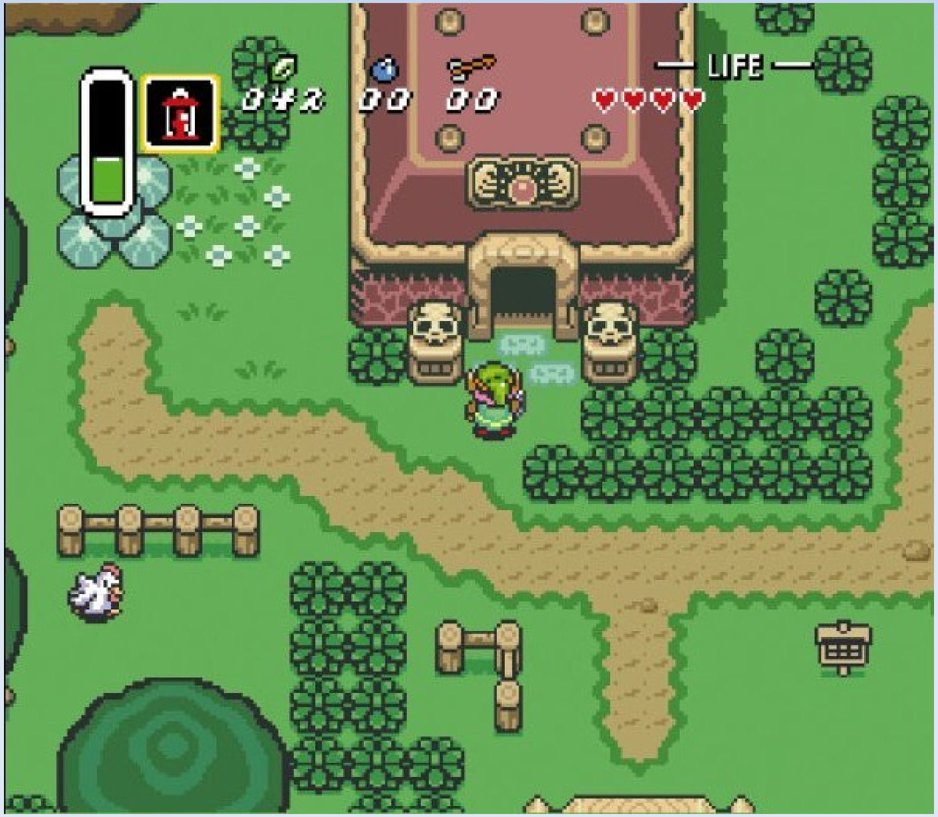
Actor Motion

- Add new state, set during motion to detect near misses
- When you release pressing the direction, clear the near miss state.
- If in a near miss state, move in the near miss direction instead

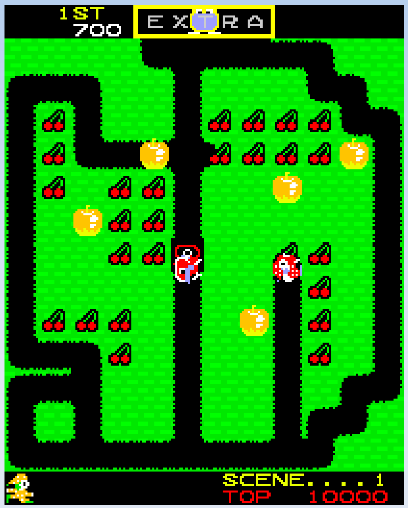


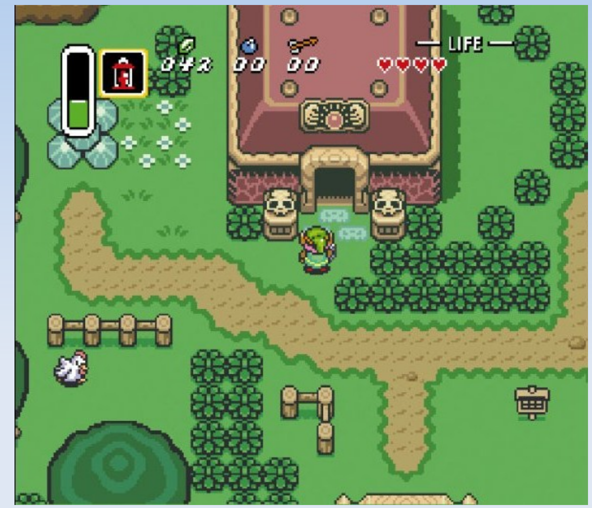
Actor Motion

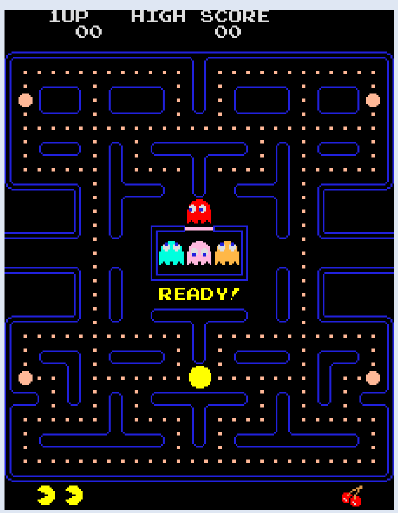
- Frame 1:
  - Link would collides with statue, set near miss left, moves right
- Frame 2:
  - Still in near miss state, move right
- Frame 3:
  - No longer in near miss state, move up



Actor Motion





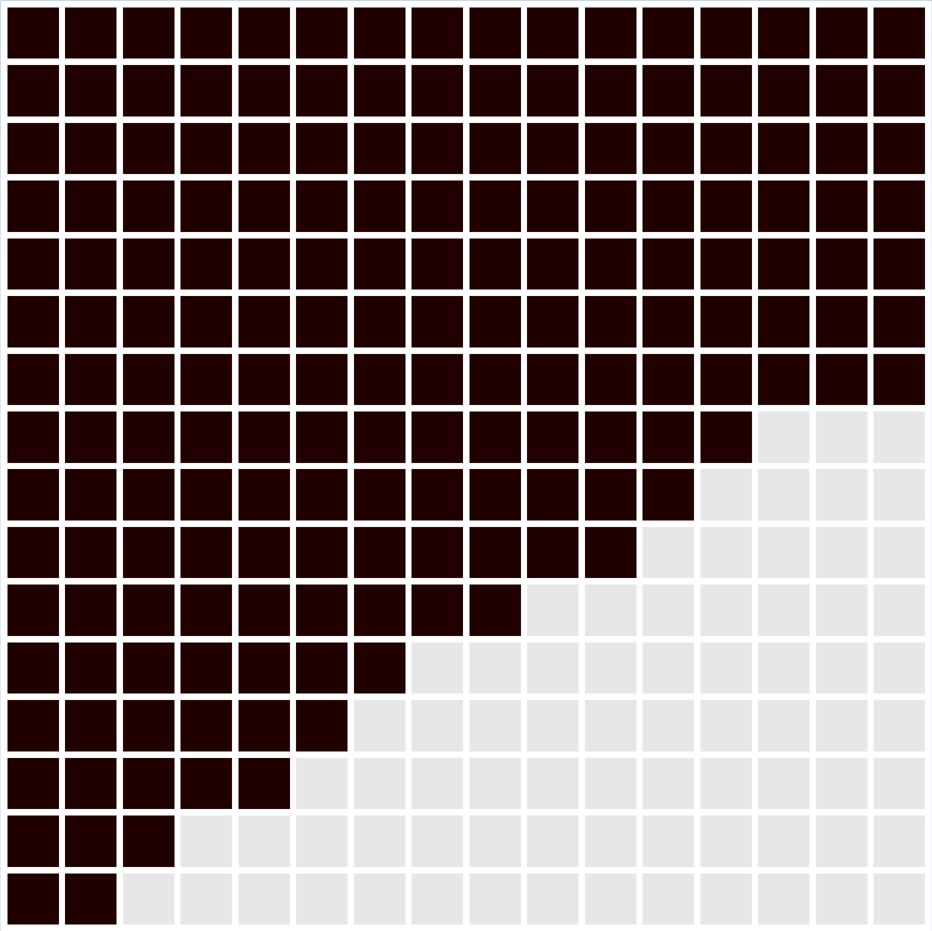


Super Mario World Example

Let's watch Super Mario World and list out all the different behaviors

Pixel Perfect Collision

- Similar techniques can be used to have arbitrary shaped platforms
- Or to create visual collision boxes



Pixel Perfect Collision

- For this to work, you need to have a 2D grid of collision information.
- Should generally line up with the art
- Can get this from the alpha channel of your images!

Pixel Perfect Collision

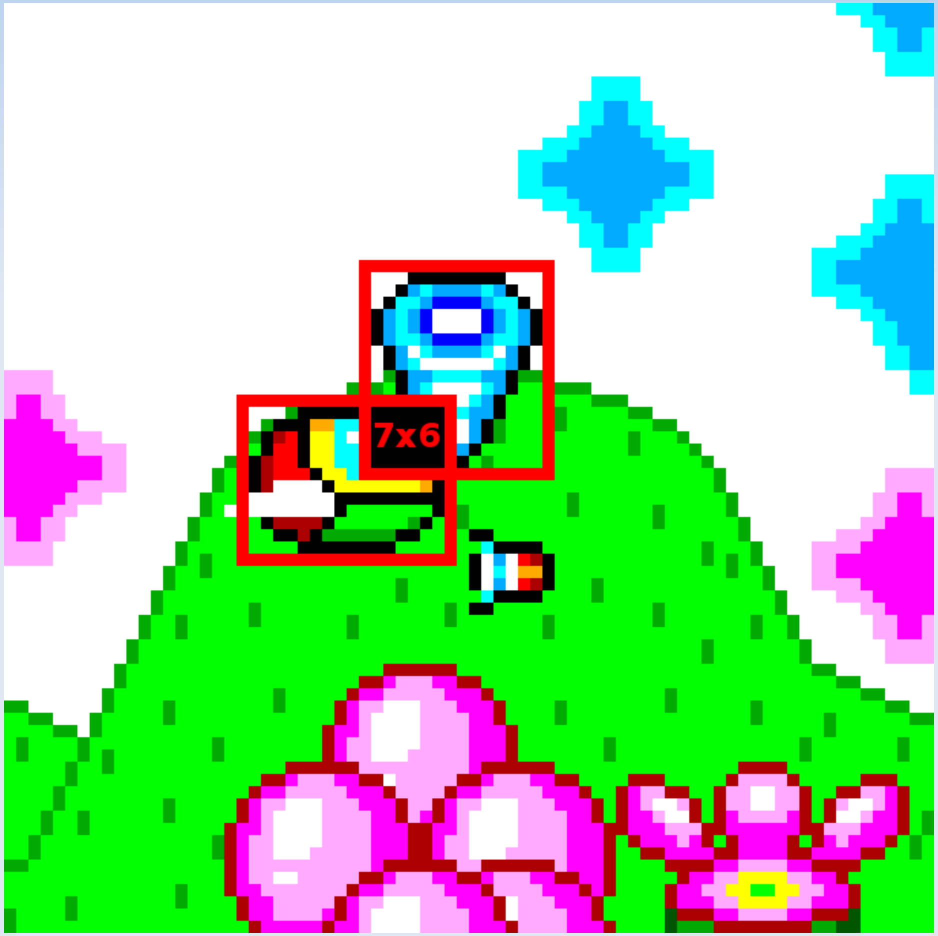
```
boolean[][] collision = new boolean[width][height];

// Read in data.
if (bitCount == 32) {
  for (int it = 0; it < imageWidth * imageHeight; ++it) {
    bytes[it * BPP + 0] = file.readByte();
    bytes[it * BPP + 1] = file.readByte();
    bytes[it * BPP + 2] = file.readByte();
    bytes[it * BPP + 3] = file.readByte();

    // Also record the alpha being zero or non-zero
    boolean isNonZero = (bytes[it * BPP + 3] != 0);
    collision[it % width][it / width] = isNonZero;
  }
} else {
```

Pixel Perfect Collision

- New collision function:
- For each pixel in object 1:
  - Find corresponding pixel in object 2
  - If both pixels are set
    - Collision
- No Collision

Corresponding Pixels

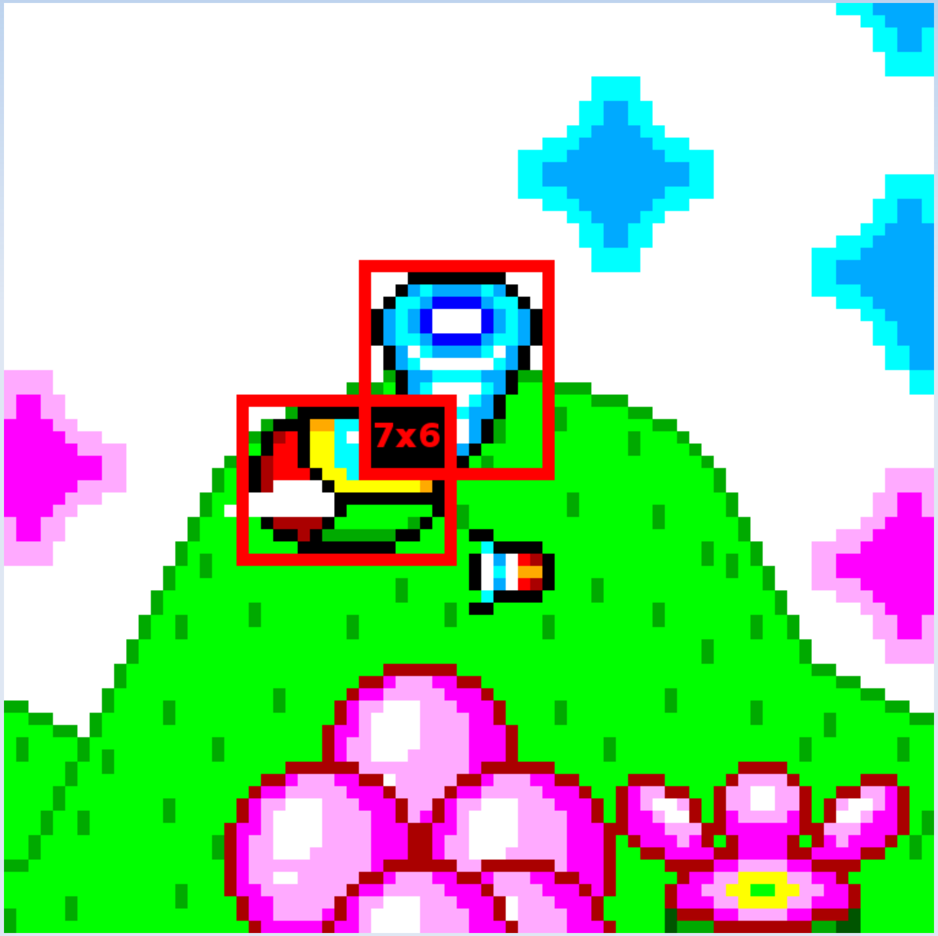
- Spaceship = 16x12
- Tornado = 14x16
- Intersection = 7x6
  - Can be calculated from AABB intersection



- T_x = S_x – (16 – 7)
- T_y = S_y + (16 – 6)

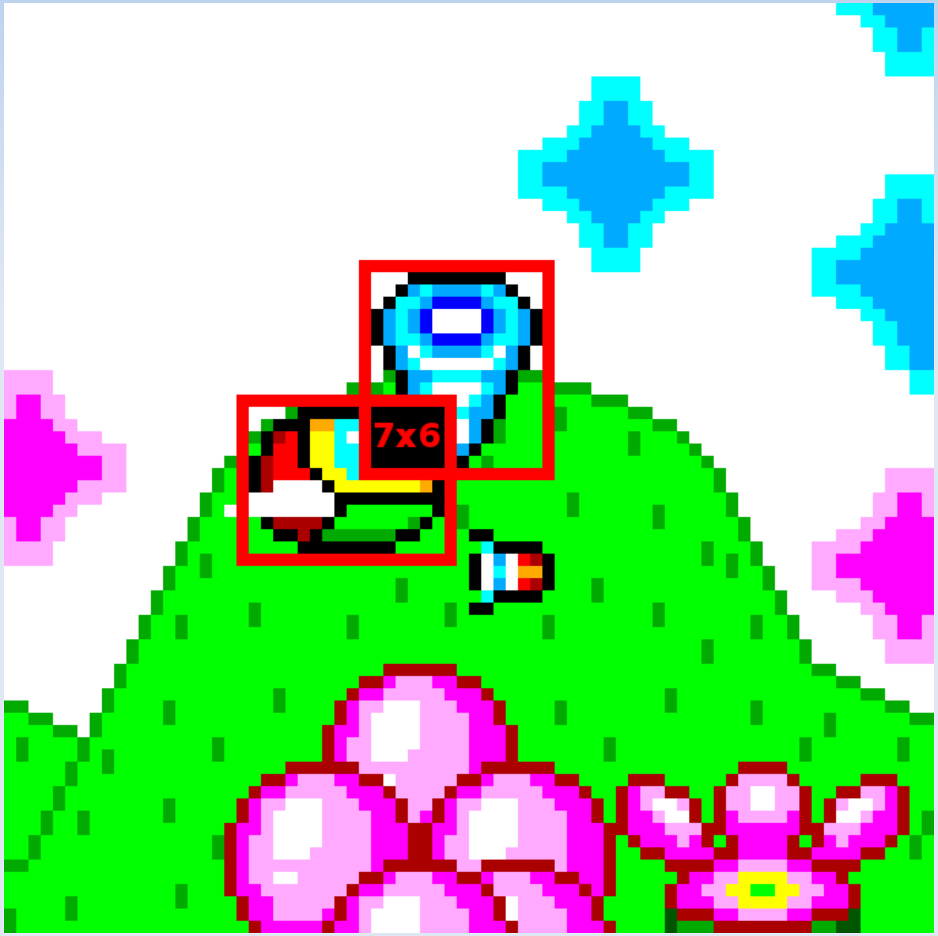
Corresponding Pixels

- $T\_x = S\_x - (16 - 7)$
- $T\_y = S\_y + (16 - 6)$
  - Y goes down!
- You SUBTRACT the overlap amount if the second sprite is greater than the first sprite.
- And ADD if the second sprite is less.



Corresponding Pixels

- For each pixel in S, calculate T:
  - $T\_x = S\_x - (16 - 7)$
  - $T\_y = S\_y + (16 - 6)$
  - If T_x or T_y is inside the tornado array AND is collidable
    - Collision
- No collision



Optimization

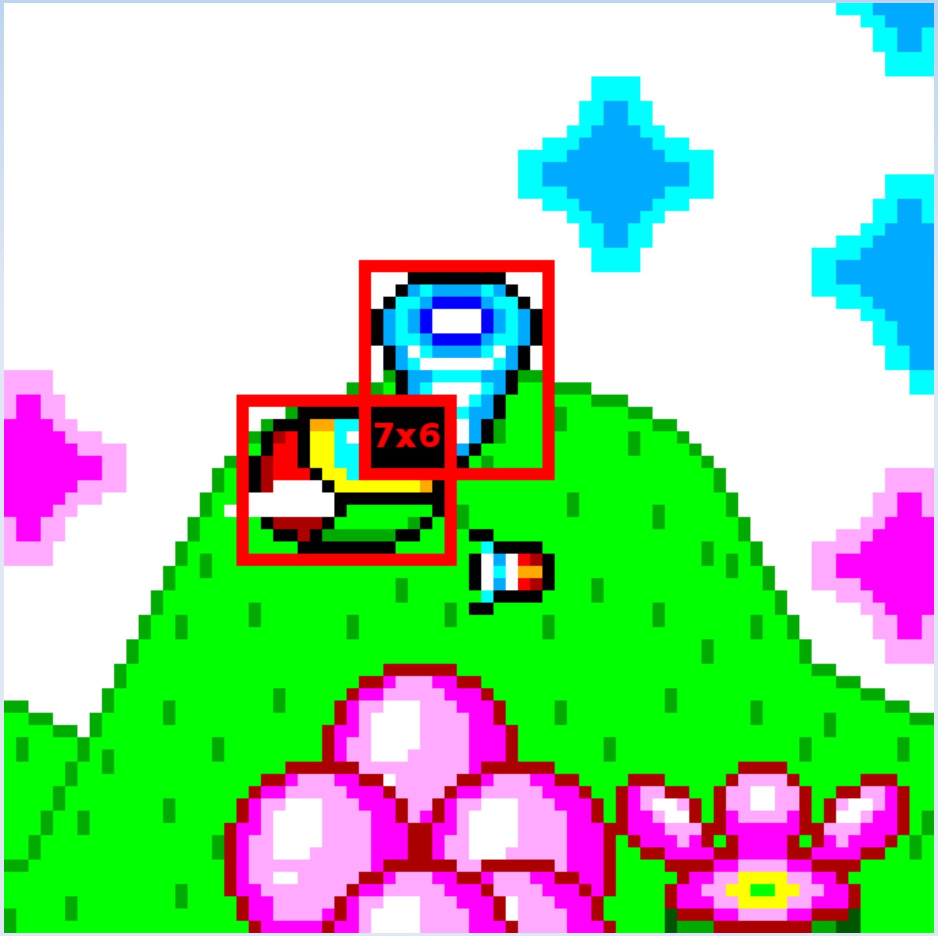
- This algorithm works, but is very very slow
  - Pixels are tested even if objects don't overlap at all!
  - Lots of pixels are tested that have no corresponding pixels in other image
  - Each individual pixel is tested as a separate operation

Corresponding Pixels

- This algorithm works, but is very very slow
  - Pixels are tested even if objects don't overlap at all!
      - Do AABB test, then do pixel test
  - Lots of pixels are tested that have no corresponding pixels in other image
      - Only test pixels in the AABB intersection
  - Each individual pixel is tested as a separate operation
      - Use bitwise arithmetic to test many pixels in parallel

Only Test Intersecting Pixels

- Arbitrarily choose one object as A, one as B
- If A_left < B_left
    - Start at A_right – intersection_w
- Else
    - Start at A_left
- Test intersection_w times



- Same thing in Y

Bitwise Operators (Java)

- & "and" Both bits must be set
- | "or" Either bit must be set
- ^ "xor" Exactly one bit must be set
- ~ "not" 1 → 0, 0 → 1
- << "lshift" mnopqrst → nopqrst0
- >> "rshift" mnopqrst → mmnopqrs
- >>> "logical rshift" mnopqrst → 0mnopqrs
  - In C, there is just >>, and its results are implementation defined (usually to be the same as in Java)

Bitwise Operators

- Use bitwise operators to test up to 32 pixels at once!
- Make bitmap literally a map of bits
  - 00011111111000→07F8
  - 00111111111100→0FFC
  - etc.
- Use bitwise AND, LSHIFT on numbers

Revised Algorithm

- Choose left, right sprites
- For each line in intersection of leftsprite:
  - (leftsprite << leftsprite_w – intersection_w) & rightsprite
  - Tests all pixels in one line at once!

- 32x – 64x faster!
- Added one limitation: sprite can not be more than 32 or 64 pixels wide

Pixel Perfect Limitations

- Pixel shape changes from frame to frame
  - Only useful when interpenetration is okay
- No information about “how far” in you are
  - Resolution must not care about that
- Best applications are where resolution is either “destroy one object” or “hurt one object then make it invincible”
  - 2D shooters, fighting games

Pixel Perfect Collisions

Questions?