# Bridges To Computing

General Information:

- This document was created for use in the "Bridges to Computing" project of Brooklyn College.
- You are invited and encouraged to use this presentation to promote computer science education in the U.S. and around the world.
- For more information about the Bridges Program, please visit our website at: http://bridges.brooklyn.cuny.edu/

# Introduction to Game Programming & Design

Lecture 3: Game State and Game Mathematics

# Content

## Player Expectations

- In general, games are held to a higher standard than other types of programs.
- People expect "office applications" to fail, and don't expect 100% up-time from business websites.
- What we are willing to tolerate when "working" is wildly different then what we are willing to tolerate when "playing".

## Efficiency

- Complexity and computability are concepts that are not normally taught on an undergraduate level.
- BUT game programmers need to consider "efficiency" in everything that they do.
- Studies have shown that if a player has to wait more than 30 seconds for levels to load in a game (3 sec on handheld devices), their "review" of that game will be greatly reduced.

## Game Mathematics

- "Game Mathematics" refers both to areas of general mathematics (geometry, trigonometry, calculus) as well as specialized areas of mathematics (vectors, matrices).
- Graphic libraries, game libraries, 2D and 3D libraries exist for programming languages to help simplify the mathematical problems that you will face. But they can't be relied on to do everything.
- Game players will not tolerate a slow game or a game that crashes. Graphics are expensive (computationally) and prone to bugs.

# Collision Detection

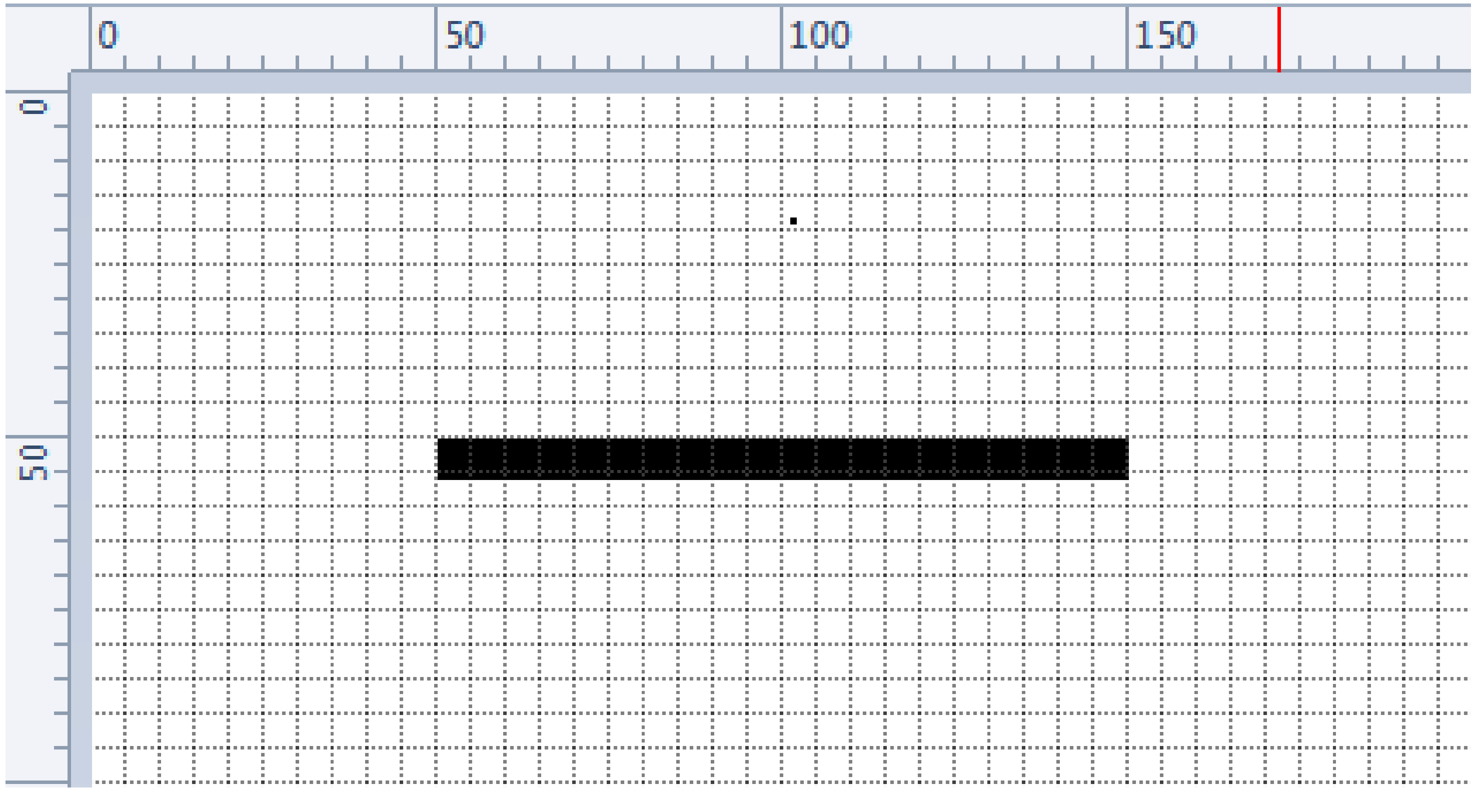
- Figuring out if two objects are touching is an incredibly common problem in a game:
  - Ball games (pong)
  - Shooting games.
- Two basic techniques:
  - <u>Overlap testing</u>
    - Detects whether a collision has already occurred
  - <u>Intersection testing</u>
    - Predicts whether a collision will occur in the future

## Overlap Testing

- Facts:
  - Most common technique used in games
  - Exhibits more error than intersection testing
- Concept
  - For every simulation step,
    1. Move (update) all objects.
    2. Test objects to see if they now overlap.
    3. If objects overlap, make adjustments or corrections.
  - Easy for simple volumes like dots, boxes and spheres, but harder for polygonal models.

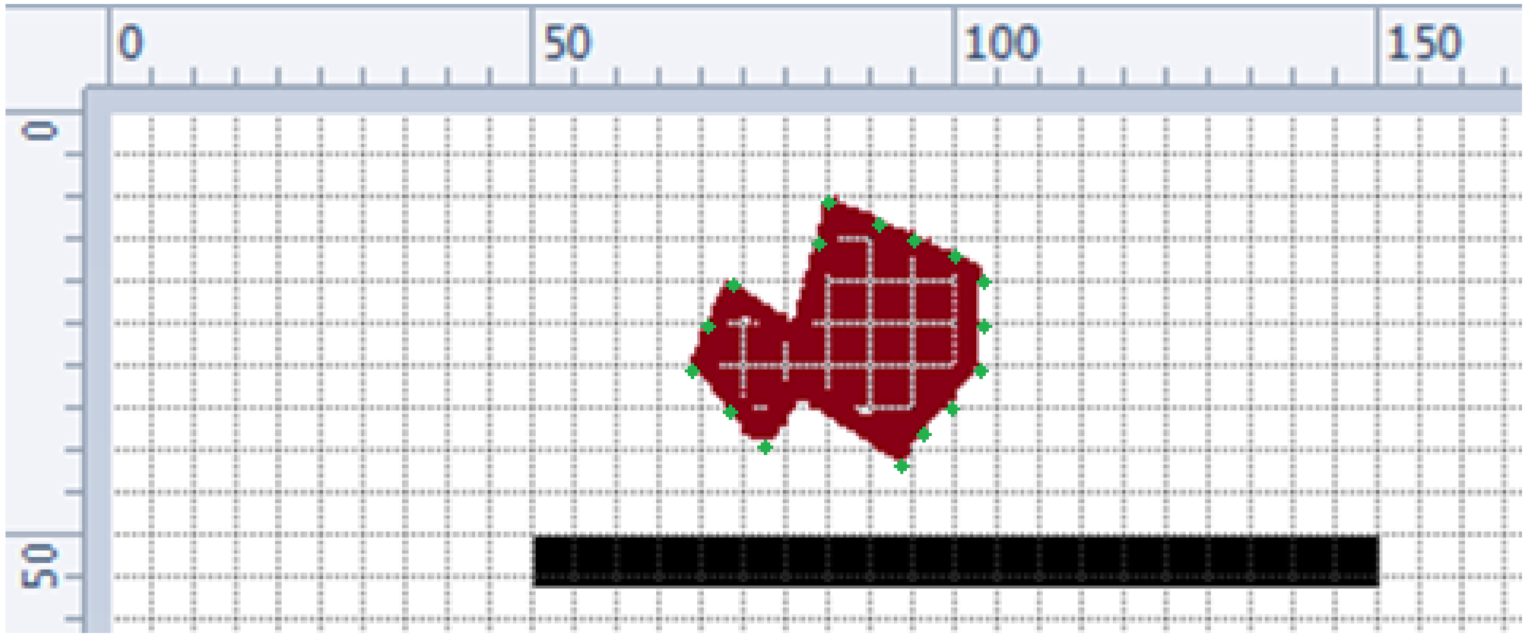
# Simple Overlap Testing

- Easiest example is of a particle interacting with a square.
- This will still require 4 logical tests in a 2D game.
- Depending on the type of game that played, <u>the order of those 4 tests can have a profound effect on efficiency</u>.



# Complex Shape OOT

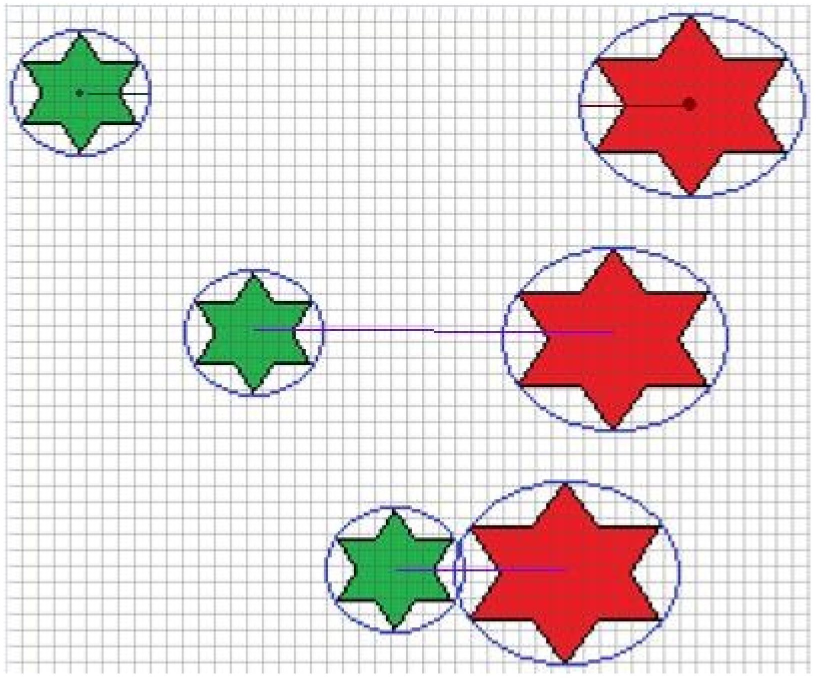
- How many tests would be required now?



- We should get <u>fairly good</u> results just testing the 16 green points against the box (4 tests each, 64 total tests)

# Bounding Circles

- For some complex shapes we can use "bounding circles": circles centered on the object that enclose all (or nearly all) of the object.
- To test whether or not two objects are colliding we can then just compare the distance between the centers of the circles to the sum of the radii of the circles (1 test).
- Bounding circles can be smaller than the object the enclose to help reduce error.



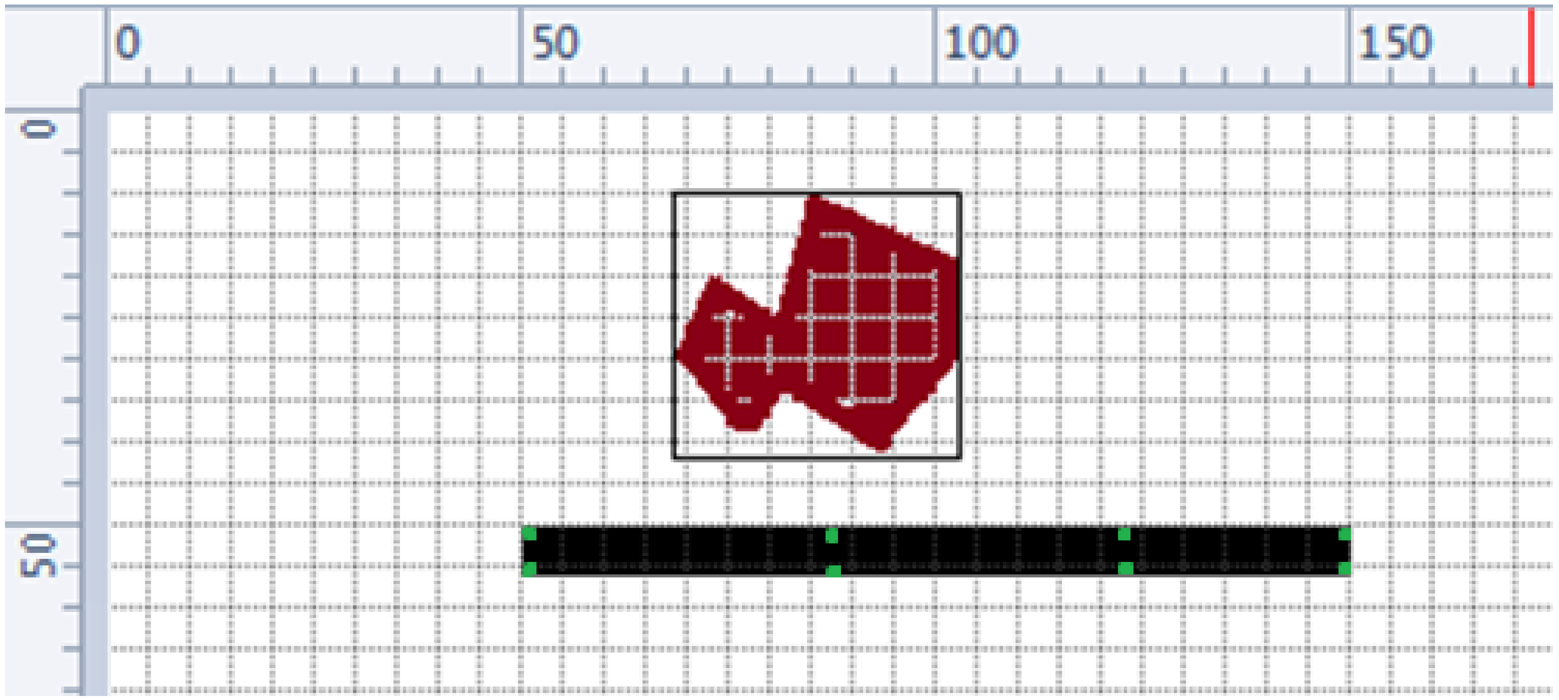
Green Star bounding circle radius = 4.5
Red Star bounding circle radius = 7

If distance between center of stars is less than 11.5 stars can be assumed to be touching. We can find the distance between the center of the stars using the Pythagorean theorem.

This will NOT give "pixel perfect" accuracy, but is usually "good enough".

# Bounding Boxes

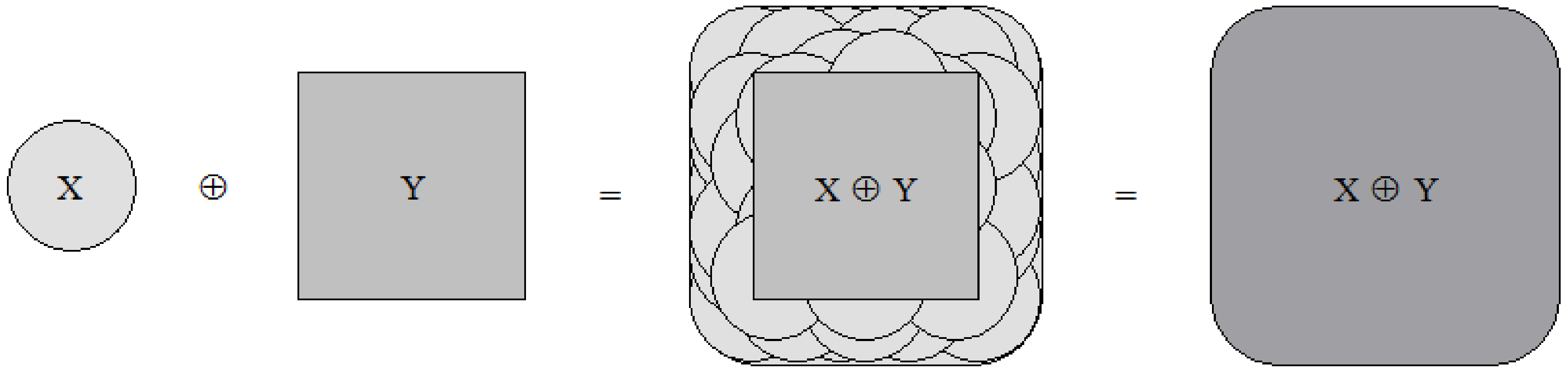
- Some long and/or thin shapes don't lend themselves to bounding circles, we can then use Bounding Boxes instead.
- Bounding Boxes can be used to reduce the complexity of shapes to simplify overlap testing.
- Note that secondary testing may need to be done if the bounding box is found to overlap.
- How many test now? (Hint: green dots)

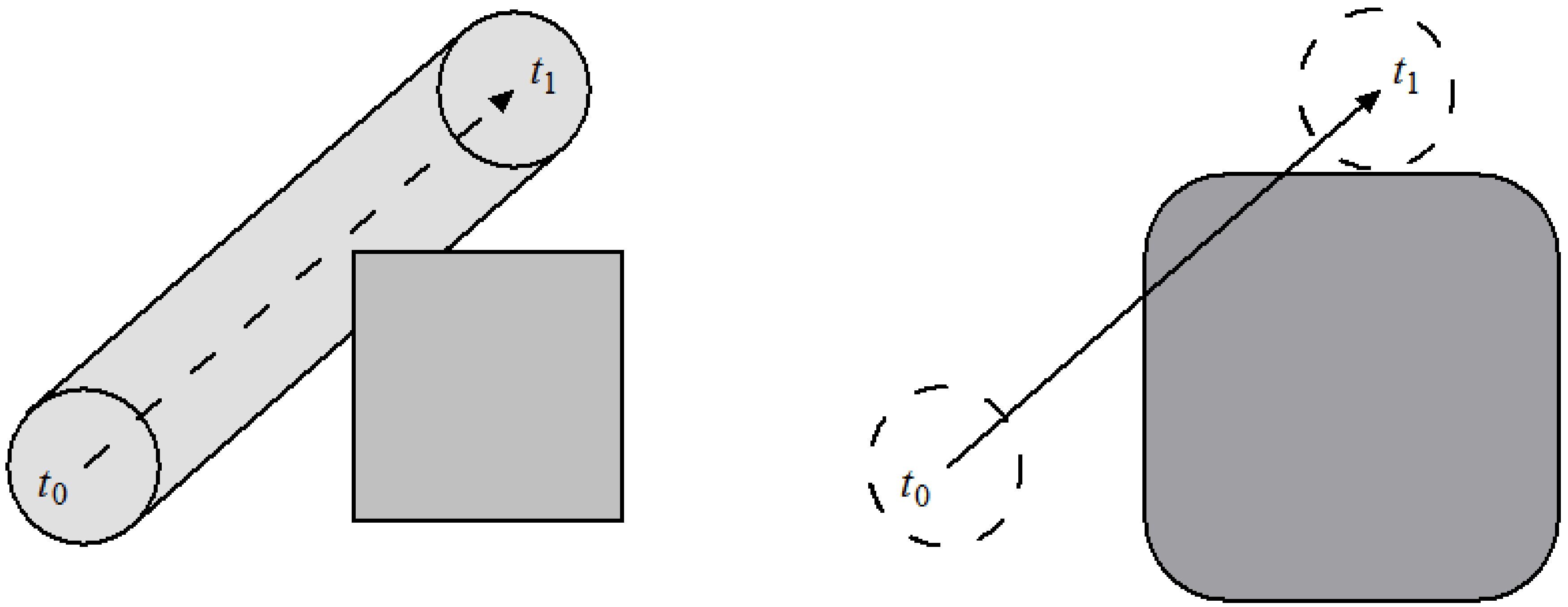


# Minkowski Sum

- In our previous example we will still need about 32 (8 dots *4) tests to determine if the boxes overlap.
- There exists a simple technique for reducing the number of tests necessary even further.
- By taking the Minkowski Sum of two complex volumes and creating a new volume, overlap can be found by testing if a single point (the center of one of our shapes) is within the new volume (4 tests).
- Variations on the Minkowski Sum include calculating the x, y and z distances between the two objects that are being tested.

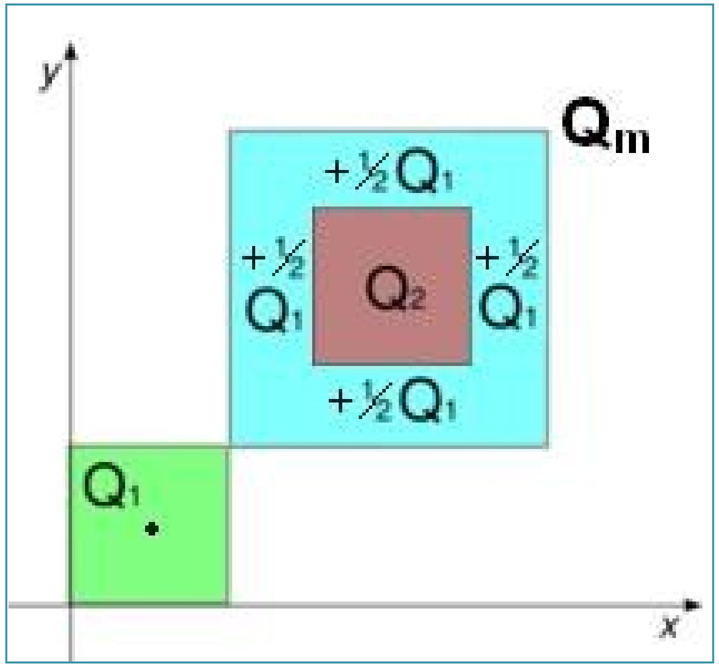
# Minkowski Sum





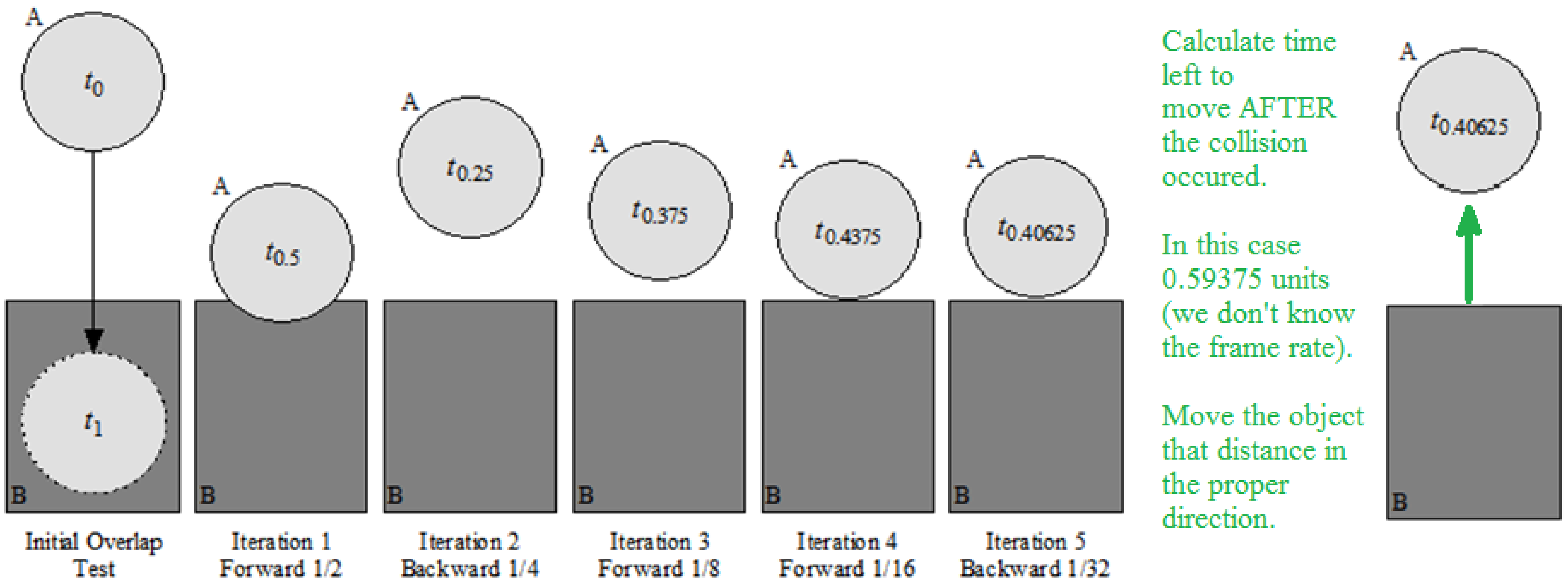
# Minkowski Sum for bounding boxes

- Find the center point of the two bounding boxes.
- Add ½ the height of the 1st boxes height to the top and bottom of the 2nd box.
- Add ½ the width of the 1st boxes width to each side of the 2nd box.
- This new box you have created is Qm (Q1 on Q2).
- Test the center of Q1 against Qm (4 tests).



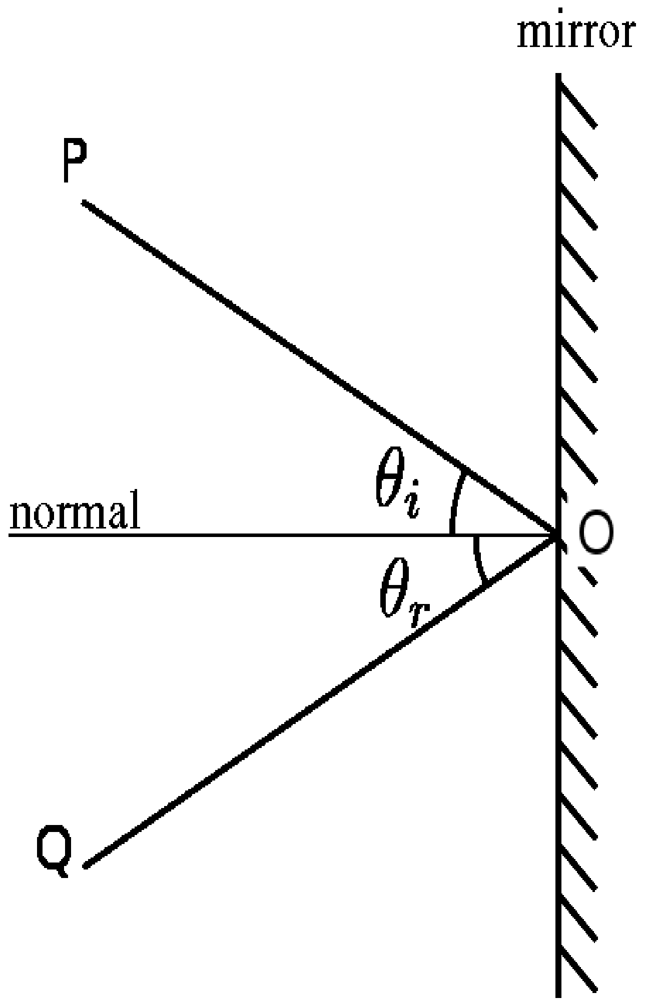
# OT – Determining Collision Time

- If we detect overlap we need to "go back" to the moment of time immediately before the objects overlapped; then calculate where the object should now be.
- Collision time is calculated in OOT by moving the object backward until right before the collision occured.
  - Bisection is one effective technique. Θ(log n)
  - Storing Minkowski values to have a standard "collision distance" another.



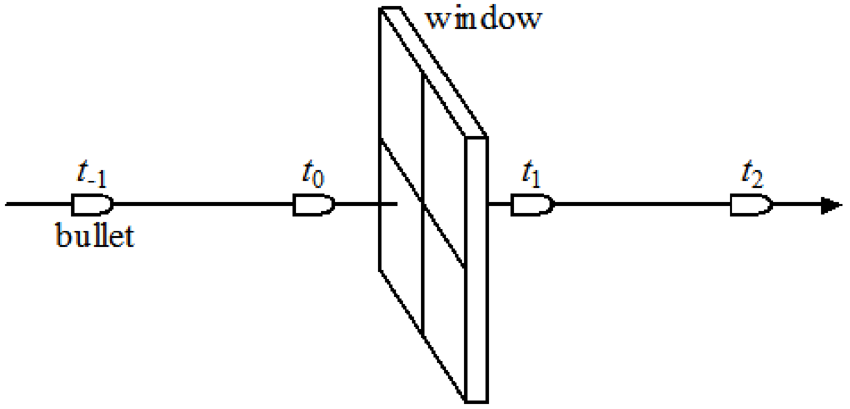
## Collision Response

- Having captured the exact moment and position of collision (O), and determined how far the object has left to move (Q)
- We can then use geometry, and trigonometry to calculate the objects resulting trajectory ( $\Theta_r$ ).



# Limits of OT

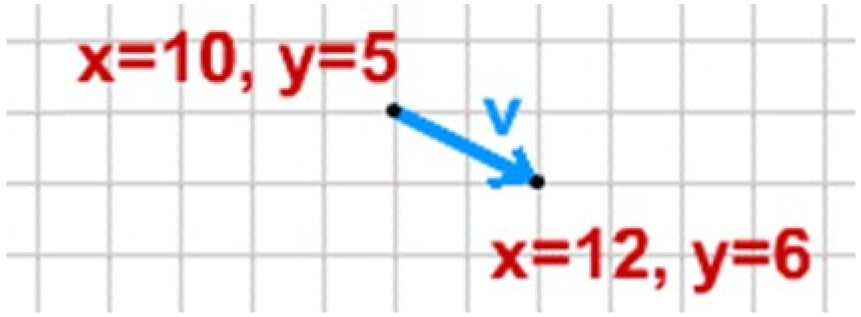
- OT is relatively easy and fast but limited.
  1. Difficult with very complex shapes
  2. Fails with objects that move too fast
     - Unlikely to catch time slice during overlap
- Possible solutions
  1. Accept that "pixel perfect" accuracy usually not necessary
  2. Constrain size/speed of objects or system:
     - Make objects really big ( halo sniper bullets... really long )
     - Design constraint on speed of objects
     - Reduce simulation step size (test physics more often than graphics)
- If none of those solutions work .. use Vectors



## Vectors

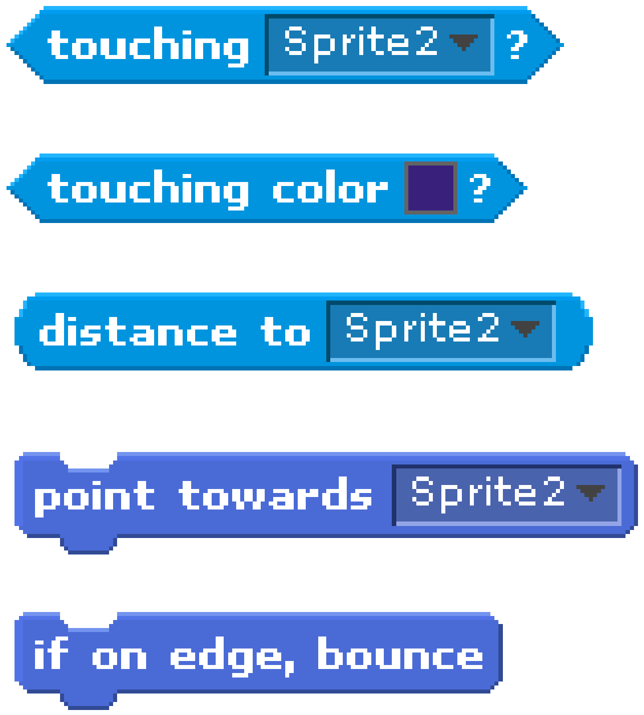
- "Vector images" are images created by a math formulas.
- We can represent  objects (and their velocity) with formulas
- Special vector mathematical operations can then be applied to reveal information about where objects will be and whether or not they will collide (at any time, past or future).
- Vector CD/CR (collision detection / collision response) is costly and slow.
- We won't be using Vectors in this class OOT is good enough.

Pv[ 10,  5, 12,  6];      // A particle vector.



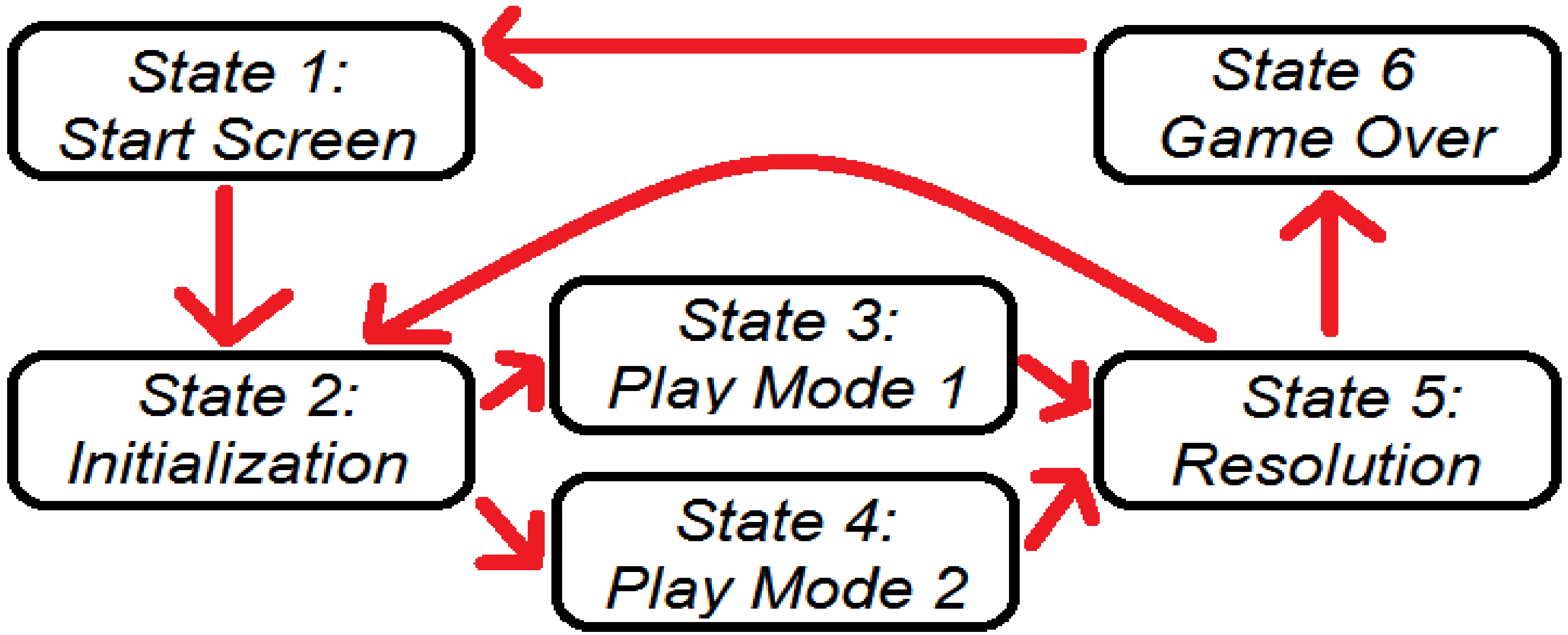
# Game Mathematics In SCRATCH

- But you won't even need to use OOT!!!
- SCRATCH simplifies game mathematics for you, with a couple of handy blocks.
  - Object collision detection in SCRATCH can be done with a simple "touching" block.
  - Objects can be kept on screen with a simple "if on edge, bounce" block.
  - Most other functions we would usually need Trig or Geometry for (point towards) also have simple solutions.



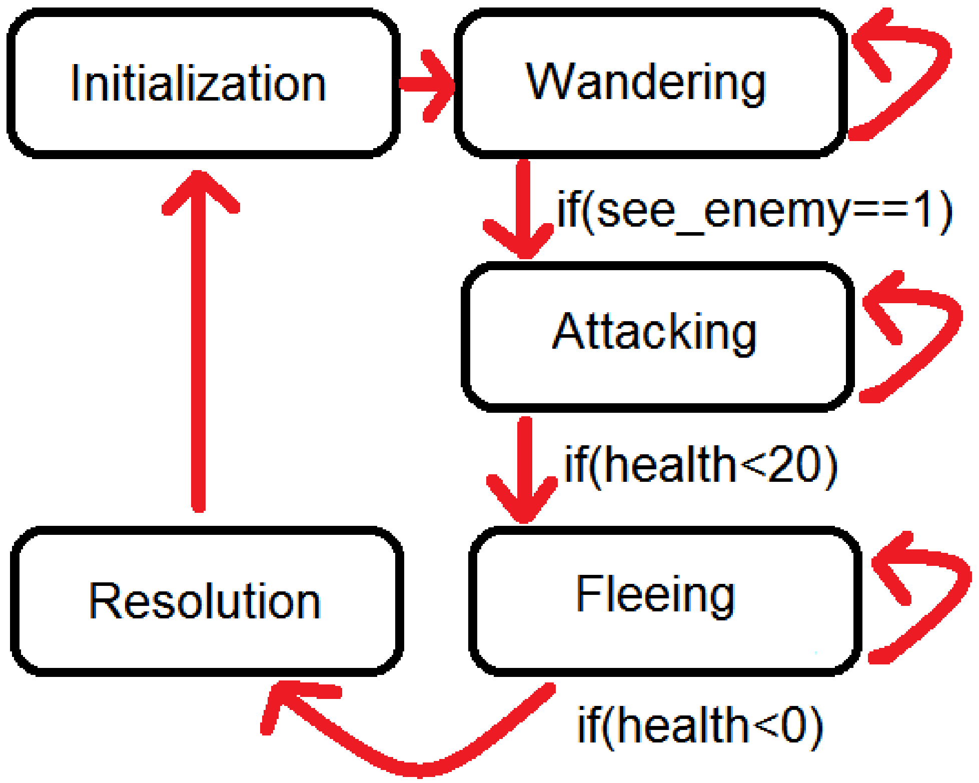
# Game State

- All games consist of a sequence of states.
- Each state is characterized by a combination of visual, audio and/or animation effects, as well as a set of rules that are being applied.



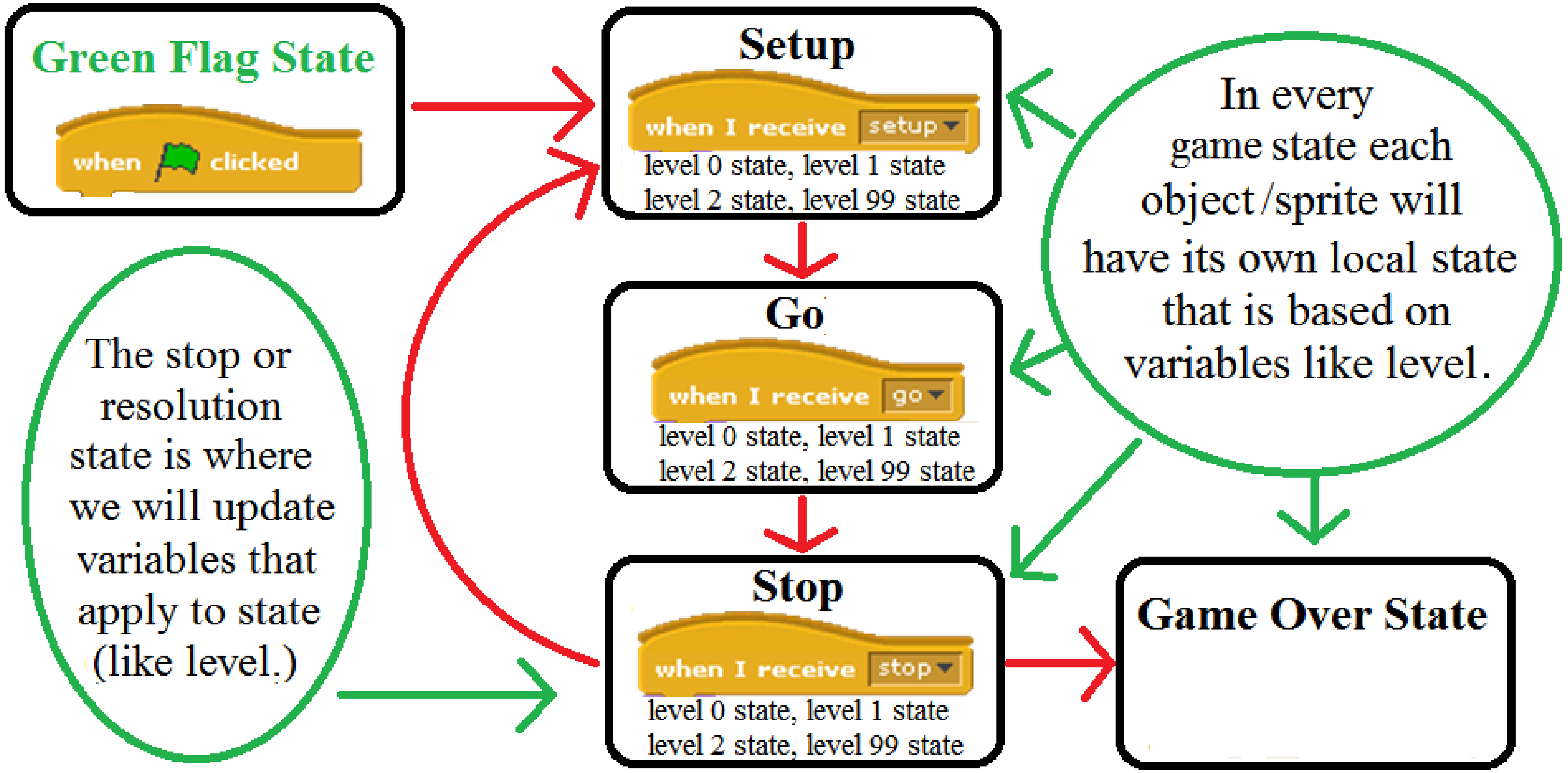
# Object State

- Objects in the game proceed through their own states as well.
- These states are defined by the behavior and functionality applied at that time.



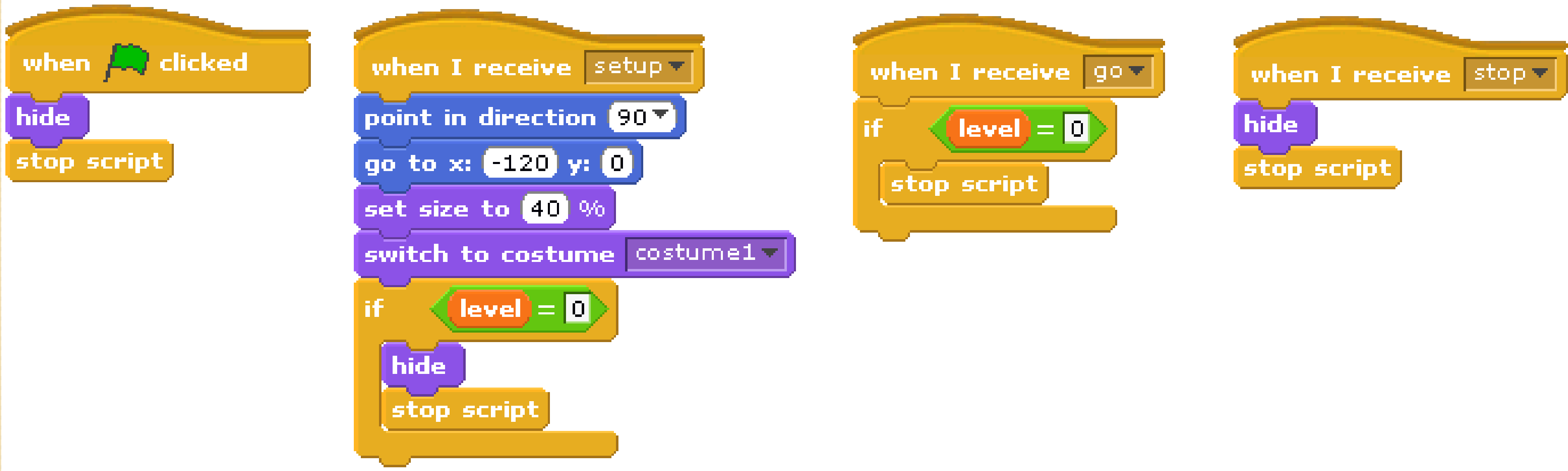
# Game State in Scratch

- At typical game in Scratch might use the following state transition diagram.


Green Flag State
when clicked
Setup
when I receive setup
level 0 state, level 1 state
level 2 state, level 99 state
Go
when I receive go
level 0 state, level 1 state
level 2 state, level 99 state
Stop
when I receive stop
level 0 state, level 1 state
level 2 state, level 99 state
Game Over State
In every game state each object/sprite will have its own local state that is based on variables like level.
The stop or resolution state is where we will update variables that apply to state (like level.)

# Object State in Scratch

- In a typical game in Scratch, all of your normal sprites (not stage or any control sprites like buttons) will work very well with only 4 scripts.
- These scripts (and any associated variables) will control the objects state.



```
when [flag] clicked
hide
stop script
```

```
when I receive setup
point in direction 90
go to x: -120 y: 0
set size to 40 %
switch to costume costume1
if level = 0
    hide
    stop script
```

```
when I receive go
if level = 0
    stop script
```

```
when I receive stop
hide
stop script
```

## The End