Collision Detection (in 2D)

Virtually all 2D action games involve collisions between objects

Whether in 2D or 3D, should distinguish *collision detection* and *collision response*

Deciding response always depends on type of material

- rigid objects that bounce – billiard ball, bullet
- rigid immovable objects – ground, walls, floor
- deformable objects that can be broken and/or or bent out of shape – balloon, skin
- insubstantial objects that offer no resistance – cloud, doorway, fire
- objects with special gameplay physics – jump-through platform

Collision Detection questions

Collision detection should be able to provide data for calculating collision response

Have two entities collided between the last frame and the present one?

And if so

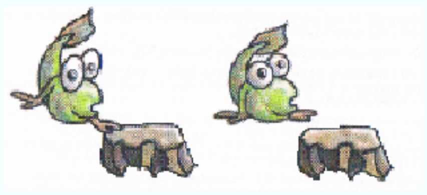
- **when** (i.e. what moment [maybe between frames]) did the collision occur?
- **where** were the entities when the collision occurred?
- which parts of the entities were in contact when the collision occurred?
  - **what is the direction of relative motion**? (for reaction bounce, for friction)

Sprite-based “exact” collision detection

Sprites have transparent pixels, visible pixels
They overlap if (after scaling, translation, rotation) they share visible pixels

Advantage: This gives precise detection of collisions, *but*

- it is an expensive operation to perform (especially if scaled and/or rotated)
- collisions are in practice relatively rare and that expense may get wasted
- precision may not even be what is wanted, esp. if collision response changes sprite
  - it may cause new sprite to have no collision, change back to old, . . . and loop



Example from “The Game Maker’s Companion”, Habgood, Nielsen & Rijks, 2010

Multi-phase collision detection

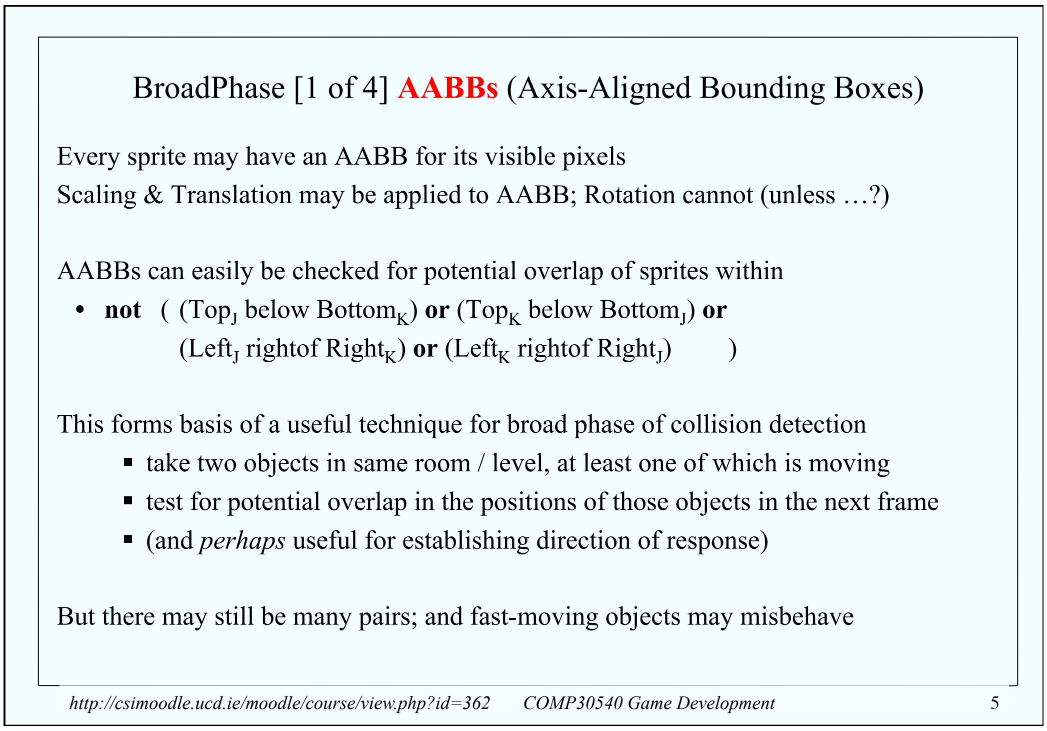
A game may contain large numbers of objects (and their sprites)
O(N^2) complexity of sprite/sprite comparisons – each of (w*h) complexity – v. costly
Collision detection should be done each frame

Simple and obvious ways to reduce the N^2 complexity

- Concentrate effort on current game level / room
- Ignore pairs of objects that didn’t move and weren’t in collision last frame

Ways to reduce the complexity of individual sprite/sprite comparison

- **Broad phase** – *quickly* eliminate pairs that *cannot possibly* be in collision
- **Narrow phase** follows – determine pairs that are *actually* in collision
  - Sometimes even acceptable to cheat and omit narrow phase! – “**colliders**”
- **Middle phase** also possible – determine pairs that *must be* in collision



BroadPhase [1 of 4] **AABBs** (Axis-Aligned Bounding Boxes)

Every sprite may have an AABB for its visible pixels
Scaling & Translation may be applied to AABB; Rotation cannot (unless …?)

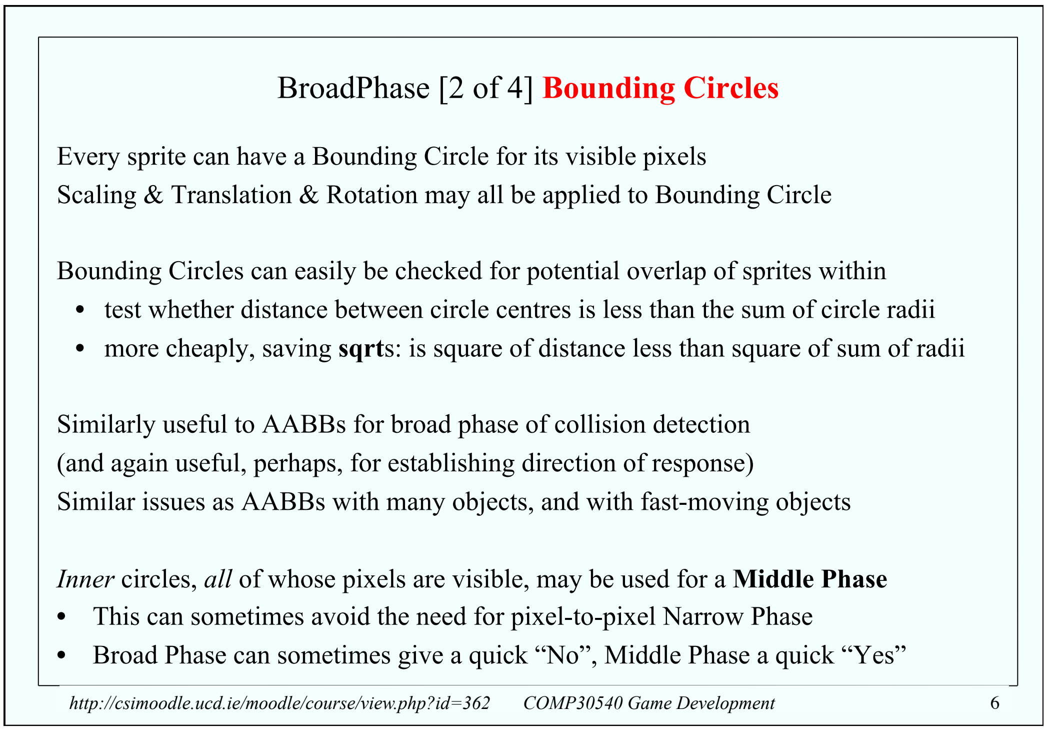
AABBs can easily be checked for potential overlap of sprites within

- **not** ((Top$_J$ below Bottom$_K$) **or** (Top$_K$ below Bottom$_J$) **or**
  (Left$_J$ rightof Right$_K$) **or** (Left$_K$ rightof Right$_J$))

This forms basis of a useful technique for broad phase of collision detection

- take two objects in same room / level, at least one of which is moving
- test for potential overlap in the positions of those objects in the next frame
- (and *perhaps* useful for establishing direction of response)

But there may still be many pairs; and fast-moving objects may misbehave



BroadPhase [2 of 4] **Bounding Circles**

Every sprite can have a Bounding Circle for its visible pixels
Scaling & Translation & Rotation may all be applied to Bounding Circle

Bounding Circles can easily be checked for potential overlap of sprites within

- test whether distance between circle centres is less than the sum of circle radii
- more cheaply, saving **sqrt**s: is square of distance less than square of sum of radii

Similarly useful to AABBs for broad phase of collision detection
(and again useful, perhaps, for establishing direction of response)
Similar issues as AABBs with many objects, and with fast-moving objects

Inner circles, *all* of whose pixels are visible, may be used for a **Middle Phase**

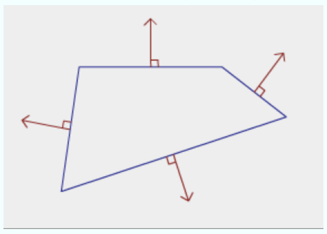
- This can sometimes avoid the need for pixel-to-pixel Narrow Phase
- Broad Phase can sometimes give a quick "No", Middle Phase a quick "Yes"

BroadPhase [3 of 4] **Separating Axis Theorem** (**SAT**)

Convex polygons do not overlap if any separating axis can be found,
ie there is some vector onto which the projections of the polygons do not overlap

AABBs are a very simple example of convex polygons.

Not all the infinitely-many possible axes need be tested: just the normals suffice



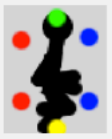
from metanetsoftware.com/technique/tutorialA.html

There are several helpful interactive illustrations there too of projections, dealing with circular shapes, and Voronoi regions

- If *any* separating axis is found, immediately stop, knowing polygons do not overlap
- If no separating axis is found, polygons do overlap

BroadPhase [4 of 4] **Collision Points** (a good idea or not?)

from wildbunny.co.uk/**blog**/2011/12/14/how-to-make-a-2d-platform-game-part-2-collision-detection/


Points around player

If the point we check on top goes inside a solid block, we move the player downward so that the top point is just below the block it bumped into. If either right point goes inside a solid block, we move the player left until the offending point is just left of the block it bumped into, and so on...

Another benefit of detecting player collision using six points in that hexagon configuration is that if the player is jumping horizontally and the feet hit a corner, the player is automatically bumped up onto the surface; if the player falling vertically hits a corner off-center, or steps off a ledge, the player slides off away from the wall. (Try this! It feels much better than it would if the player behaved as a boxy rectangle.)

and his subsequent critique: the player being automatically bumped onto the surface was very jarring and often left me wondering what had happened while playing the game. … when the player was moving quickly and became embedded inside a block, … with all points inside the block, there was no clearly correct way to resolve. … the amount of code I found I needed was getting excessive

Fast-moving objects?

A fast object might cross right over another object in the time between frames

- Define “Fast”: it moves by at least 1/3 its smallest dimension between frames

Create AABB for each moving object encompassing its past and future positions

(This assumes straight-line movement in the time between frames: why is this reasonable?)

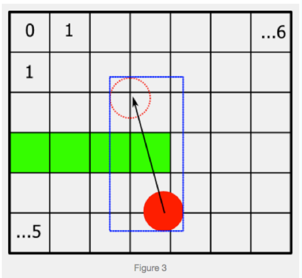


Figure 3

- In a Broad phase, this will allow potential collisions to be detected
- The Narrow phase will need to estimate positions of objects between frames
- The Broad phase may need to compare AABBs with Bounding Circles, etc

Have you got very many objects? Then use Quadtrees

Quadtrees provide a way of partitioning 2D space. Each node represents a rectangle.

Each node may have four child nodes, dividing its area in four quarters

This tree structure may repeat as many or as few depths as you require

- If tree is deep, each node covers a small area and usually contains few objects
  - but it probably contains only parts of those objects, other parts of those objects will be contained in multiple other nodes
- If tree is shallow, … large area, many objects – objects may still be spread out
- Tree depth does not in principle have to be uniform

Each leaf node has a list of *relevant objects* which occupy some of its space

Each object is placed in as many lists of *relevant objects* as necessary

Checking objects using Quadtrees

Only moving objects need to be added/removed from lists between frames

Only moving objects need be checked for collisions with other objects

Moving objects need be checked only against other objects in the same lists

Computational Tradeoff: list management versus object comparisons

- deep tree: objects typically in many lists, each such list quite small
- shallow tree: objects typically in one or a few lists, but each list may be large