CSE 3902: Collision Detection and Response

Justin Holewinski

The Ohio State University

Collision Basic Concepts

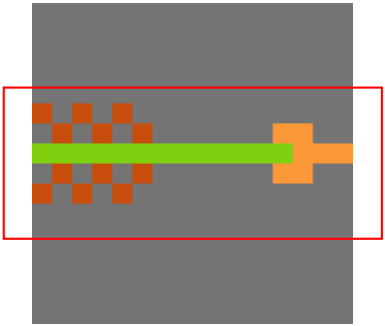
Collision *detection* is identifying if/when two objects collide.

- Is the player hitting an enemy?
- Is the player hitting a wall?
- Is an enemy hitting another enemy?

Collision *response* is the logic to *resolve* a collision.

- Prevent overlapping objects
- Damage player
- Move to next room

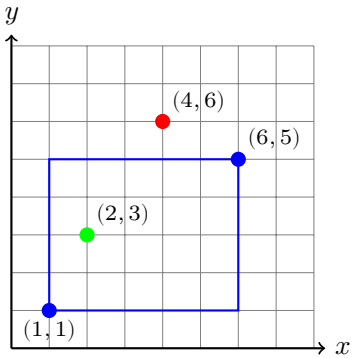
Bounding Boxes



A *bounding box* represents the collidable part of a game object

- Part of a sprite that is collidable
- Invisible trigger for level transition

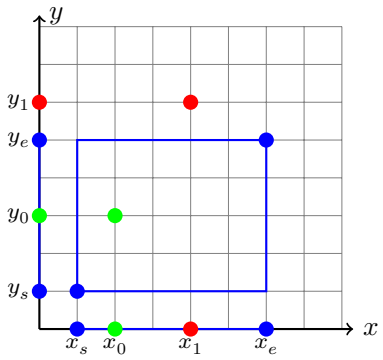
Point/Rectangle Collision



Assumption: Rectangle is *axis-aligned*

How do we check if a point is *inside* a rectangle?

Point/Rectangle Collision



Flatten coordinates on each axis
Check for overlap

Does (x_0, y_0) collide with rectangle?

$$\begin{array}{rcrcrc} x_s & \leq & x_0 & \leq & x_e & \checkmark \\ y_s & \leq & y_0 & \leq & y_e & \checkmark \end{array}$$

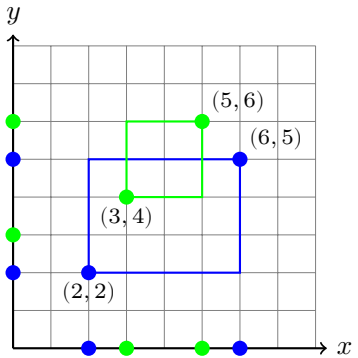
Yes!

Does (x_1, y_1) collide with rectangle?

$$\begin{array}{rcrcrc} x_s & \leq & x_1 & \leq & x_e & \checkmark \\ y_s & \leq & y_1 & \leq & y_e & \mathbf{X} \end{array}$$

No!

Rectangle/Rectangle Collision



Assumption: Rectangles are *axis-aligned*

As before, flatten onto each axis

Observation: Overlap occurs on an axis if each rectangle *starts* before the other *ends*.
If A and B are rectangles:

$$x_{A,start} < x_{B,end} \quad \wedge \quad x_{B,start} < x_{A,end}$$
$$y_{A,start} < y_{B,end} \quad \wedge \quad y_{B,start} < y_{A,end}$$

Rectangle/Rectangle Collision

MonoGame uses `Left`, `Right`, `Top`, `Bottom`:

```
// https://github.com/MonoGame/MonoGame/blob/develop/MonoGame.Framework/Rectangle.cs
public bool Intersects(Rectangle value)
{
    return value.Left < Right &&
           Left < value.Right &&
           value.Top < Bottom &&
           Top < value.Bottom;
}
```

Why does MonoGame use `Top < Bottom` instead of `Bottom < Top`?

Collision "Sides"

Collision detection in 2D games is often more than just a boolean check

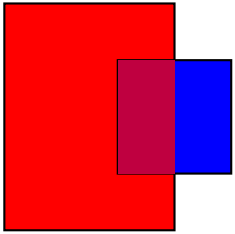
- *How* objects collide can be just as important as *if* they collide

Mario Examples

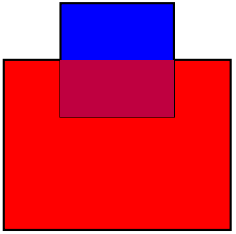
- Does Mario always take damage when colliding with a Goomba?
- Does it matter how Mario collides with a brick block?

Collision "Sides"

Left-Right Collision



Top-Bottom Collision



For square-ish objects, look at intersection region

- $width < height$: left-right collision
- $width > height$: top-down collision

Collision "Sides"

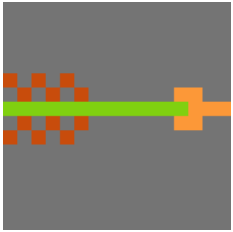
MonoGame makes it easy to determine intersection region:

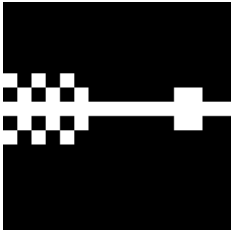
```
// https://github.com/MonoGame/MonoGame/blob/develop/MonoGame.Framework/Rectangle.cs

public static void Intersect(ref Rectangle value1, ref Rectangle value2, out Rectangle result)
{
    if (value1.Intersects(value2))
    {
        int right_side = Math.Min(value1.X + value1.Width, value2.X + value2.Width);
        int left_side = Math.Max(value1.X, value2.X);
        int top_side = Math.Max(value1.Y, value2.Y);
        int bottom_side = Math.Min(value1.Y + value1.Height, value2.Y + value2.Height);
        result =
            new Rectangle(left_side, top_side, right_side - left_side, bottom_side - top_side);
    }
    else
    {
        result = new Rectangle(0, 0, 0, 0);
    }
}
```

Pixel-Accurate Collision

Bounding boxes are (usually) not pixel-accurate; can we do better?

Generate mask of visible pixels, which can easily be obtained from alpha channel





Place masks at same location as sprites and check for overlapped non-zero mask pixels.

Per-pixel collision is *expensive*

- First locate potential collisions using bounding boxes
- Then perform per-pixel collision detection on potentials

Many-to-Many Collision Detection

Brute-force:

```
foreach (ICollidable first in World.Collidables) {
    foreach (ICollidable second in World.Collidables) {
        if (first.Intersects(second)) {
            HandleCollision(first, second);
        }
    }
}
```

What is wrong with this?

Many-to-Many Collision Detection

We can speed this up by being smarter about what we need to check

- Eliminate duplicates (e.g. `A.Intersects(B)` is the same as `B.Intersects(A)`)
- Partition entities based on what they can collide with
  - Start with *static* vs *dynamic* entities
  - Player *can* collide with enemies
  - Static blocks *cannot* collide with other static blocks
- Use *spatial partitioning* to accelerate the intersection checks

Spatial Partitioning

Observation: collision detection is only necessary if the objects are close to each other.

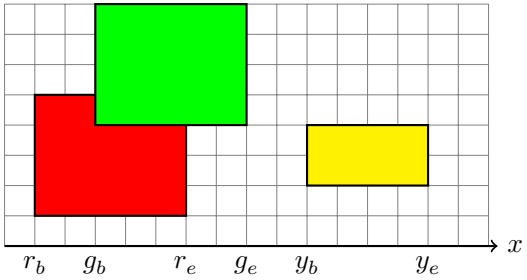
How do we define "close"?

- Partition world/level/room into disjoint areas
- If objects are in different disjoint areas, they are trivially not intersecting

Examples of algorithms:

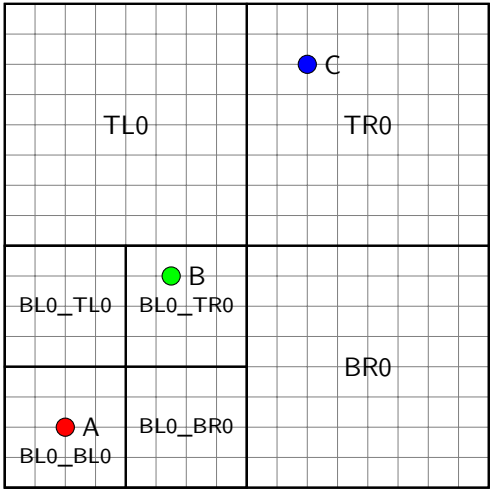
- Sort and Sweep
- Quadtrees

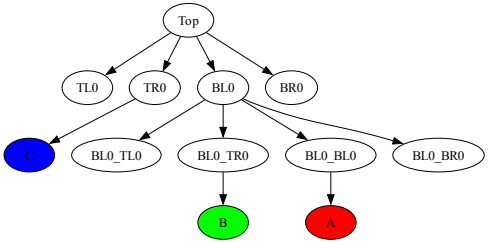
Sort and Sweep



- Sort objects along an axis
- Sweep through the axis looking for overlap
  - When you see an object start, it is eligible for collision
  - When you see an object end, it is no longer eligible for collision

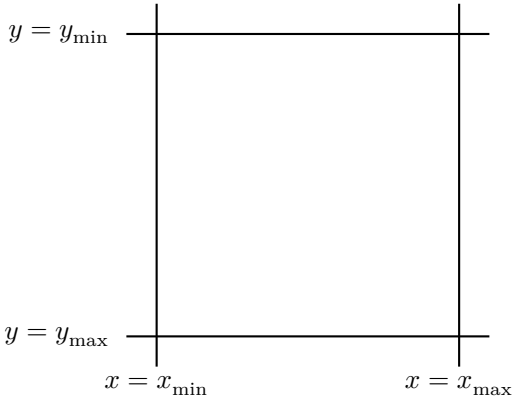
Quadtrees


C
TL0
TR0
B
BL0_TL0
BL0_TR0
BR0
A
BL0_BL0
BL0_BR0


Top
TL0
TR0
BL0
BR0
C
BL0_TL0
BL0_TR0
BL0_BL0
BL0_BR0
B
A

Level Collisions

For Zelda-like dungeon rooms, planes can be used



Level Collisions

Or use bounding box rectangles

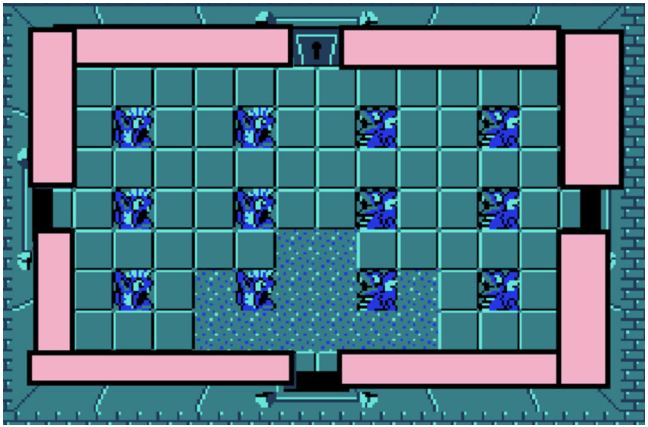


Image taken from Matt Boggus' lecture slides

2D Physics

Position: Vector from origin indicating location

Velocity: Rate of change of position

Acceleration: Rate of change of velocity

Equations of motion:

$$\begin{aligned} \vec{x}_t &= \vec{x}_{t-1} + \vec{v} * \Delta t \\ \vec{v}_t &= \vec{v}_{t-1} + \vec{a} * \Delta t \end{aligned}$$

No Acceleration

With no acceleration, position update is easy:

$$\begin{aligned}
\vec{v}_t &= \vec{v}_{t-1} + \vec{a} * \Delta t \\
&= \vec{v}_{t-1} + 0 * \Delta t \\
&= \vec{v}_{t-1} \\
\vec{x}_t &= \vec{x}_{t-1} + \vec{v} * \Delta t
\end{aligned}$$

Constant Acceleration

With constant acceleration, position update is a bit trickier:

$$\begin{aligned} \vec{v}_t &= \vec{v}_{t-1} + \vec{a} * \Delta t \\ \vec{x}_t &= \vec{x}_{t-1} + \vec{v} * \Delta t \\ &= \vec{x}_{t-1} + ??? * \Delta t \end{aligned}$$

What do we plug in for $\vec{v}$ above? $\vec{v}_t$? $\vec{v}_{t-1}$?

We can approximate with an average:

$$\begin{aligned} \vec{x}_t &= \vec{x}_{t-1} + \vec{v} * \Delta t \\ &= \vec{x}_{t-1} + \frac{\vec{v}_t + \vec{v}_{t-1}}{2} * \Delta t \end{aligned}$$

Player Movement

Input devices can change player *velocity* instead of *position*

- In sprint 2, pressing `D` may alter the player's position
- Pressing `D` may instead increase the player's velocity in the $+x$ direction

Things to consider

- How does player state affect user impact on movement?
  - Jumping
  - Damaged
- Death planes
  - What happens if the user falls through the level?
  - Kill player if $y >$ MAX_HEIGHT (remember MonoGame uses $+y$ for down)

Player Movement - Clamping

Player (usually) should not continue increasing velocity past a particular threshold, so we can *clamp* the value:

```
public void IncreaseVelocityX(float amount)
{
    Velocity.X += amount;
    Velocity.X = Math.Min(Velocity.X, MAX_VELOCITY);
    Velocity.X = Math.Max(Velocity.X, -MAX_VELOCITY);
    // OR
    Velocity.X = Math.Clamp(Velocity.X, -MAX_VELOCITY, MAX_VELOCITY);
}
```

What about operating on the velocity as a vector?

```
public void IncreaseVelocity(Vector2 amount)
{
    Velocity += amount;
    // ???
}
```

Player Movement - Decay

What happens when the user releases a movement key?

- Does the player instantly stop?

How do we gradually stop the player?

- *Decay* the velocity, gradually pushing towards zero

$$\vec{v}_t = \lambda \vec{v}_{t-1}, \quad \text{where} \quad 0 < \lambda < 1$$

```
public Vector2 ApplyDecay()
{
    Velocity = Velocity * VELOCITY_DECAY;
}
```

Collision Response - Software Organization

Where is collision response handled in your project?

- In the `Player`, `Enemy`, `Block`, ... classes?
- In the `Level` class?
- In a dedicated physics handler class?
- In handler classes for specific objects?

Collision Response in Game Object Classes

```
public class Player
{
    // ...

    public void HandleBlockCollision(IBlock other)
    {
        // ...
    }

    public void HandleItemCollision(IItem item)
    {
        // ...
    }

    public void HandleEnemyCollision(IEnemy item)
    {
        // ...
    }
}
```

`Block`, `Enemy`, `Item`, and other classes would have similar methods.

How does this affect our code quality?

- Coupling
- Cohesion
- Maintainability
- Extendability

Collision Response in Level Class

```
public class Level
{
    // ...

    public void HandlePlayerBlockCollision(IPlayer player,
                                           IBlock other)
    {
        // ...
    }

    public void HandlePlayerItemCollision(IPlayer player,
                                          IItem item)
    {
        // ...
    }

    public void HandlePlayerEnemyCollision(IPlayer player,
                                           IEnemy item)
    {
        // ...
    }
}
```

Does this help?

- Coupling
- Cohesion
- Maintainability
- Extendability

Collision Response in Dedicated Object Handler

```
public class PlayerCollisionHandler
{
    // ...
    IPlayer player;

    public void HandleBlockCollision(IBlock other)
    {
        // ...
    }

    public void HandleItemCollision(IItem item)
    {
        // ...
    }

    public void HandleEnemyCollision(IEnemy item)
    {
        // ...
    }
}
```

`Block`, `Enemy`, `Item`, and other classes would have similar handlers.

How does this affect our code quality?

- Coupling
- Cohesion
- Maintainability
- Extendability

Collision Response in Dedicated Object Pair Handler

```
public class PlayerEnemyCollisionHandler
{
    // ...

    public void HandleCollision(IPlayer player, IEnemy other)
    {
        // ...
    }
}

public class EnemyBlockCollisionHandler
{
    // ...

    public void HandleCollision(IEnemy other, IBlock block)
    {
        // ...
    }
}
```

How does this affect our code quality?

- Coupling
- Cohesion
- Maintainability
- Extendability

Collision Response Handler Management

Type	Type	Handler
`IPlayer`	`IBlock`	`PlayerBlockHandler`
`IPlayer`	`IEnemy`	`PlayerEnemyHandler`
`IPlayer`	`IItem`	`PlayerItemHandler`
`IEnemy`	`IBlock`	`EnemyBlockHandler`
`IEnemy`	`IEnemy`	`EnemyEnemyHandler`
`IEnemy`	`IItem`	`EnemyItemHandler`
...		

How could you represent this table look-up in C#?

- `if-then-else` chains?
- `Dictionary` look-up?
- Some other data structure?

Collision Response Handler Management

We could also use the *command* pattern:

Type	Type	Side	Command
`Player`	`Goomba`	`Left`	`PlayerTakeDamageCommand`
`Player`	`Goomba`	`Right`	`PlayerTakeDamageCommand`
`Player`	`Goomba`	`Top`	`EnemyTakeDamageCommand`
`Player`	`Star`	`Any`	`PlayerGetsStarPowerCommand`
`Player`	`BrickBlock`	`Bottom`	`BreakBlockCommand`
`Player`	`Block`	`Any`	`ResolvePlayerOverlapCommand`
...			

Again, how could you represent this table look-up in C#?

- `if-then-else` chains?
- `Dictionary` look-up?
- Some other data structure?