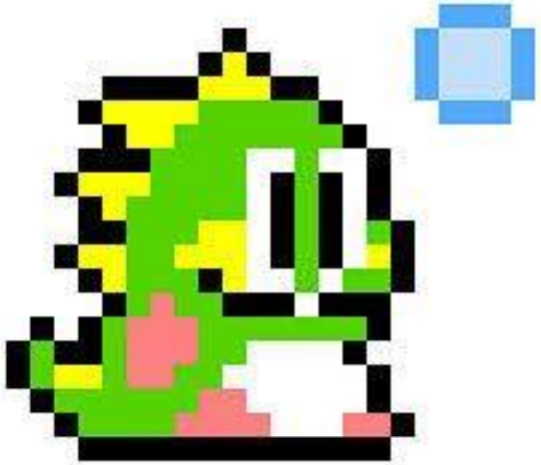
# Game Programming

## Lecture 2: Game Objects

Hierarchy, Collisions, Interactions

December 11, 2023

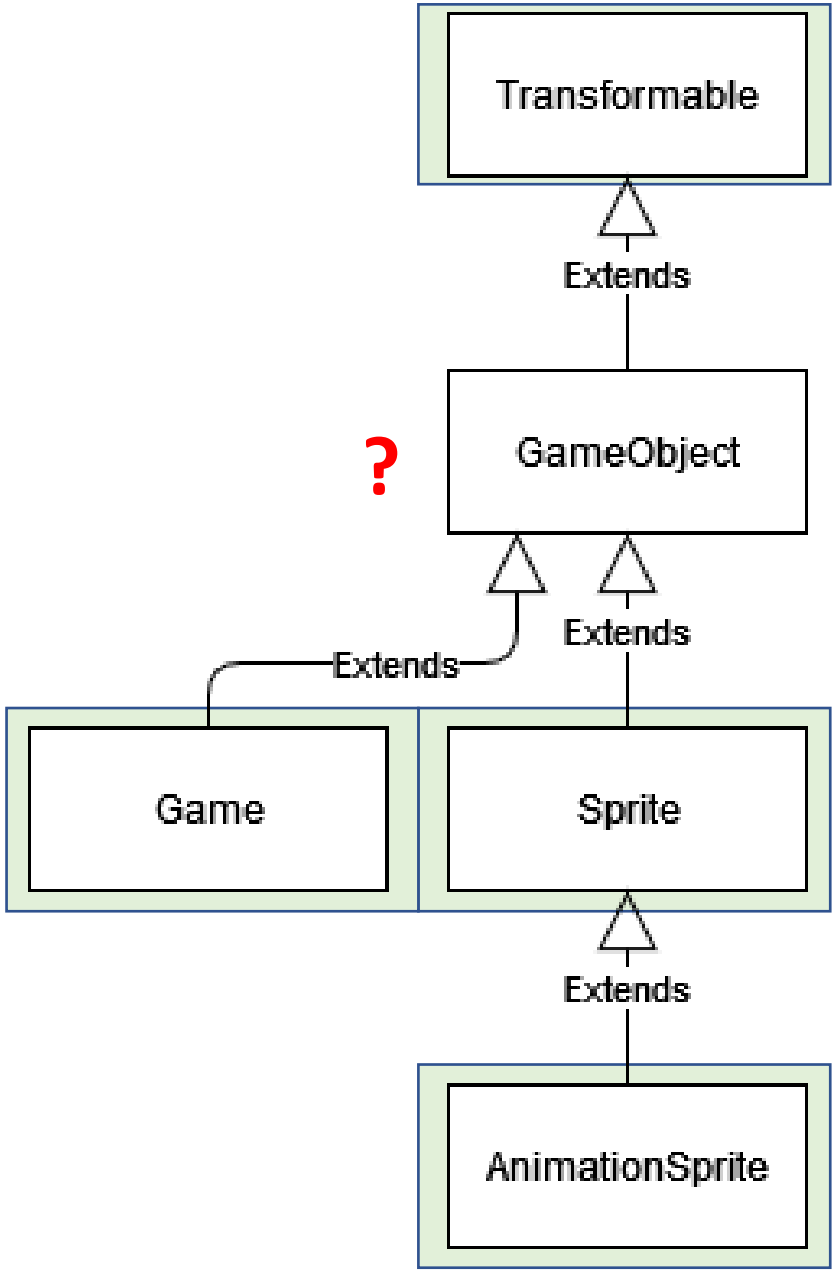
Paul Bonsma



# Last Week

- Getting started with:
  - C#
  - IDE (Visual Studio)
  - GXPEngine
- Task: Creating an animated player object that can be moved using keyboard input

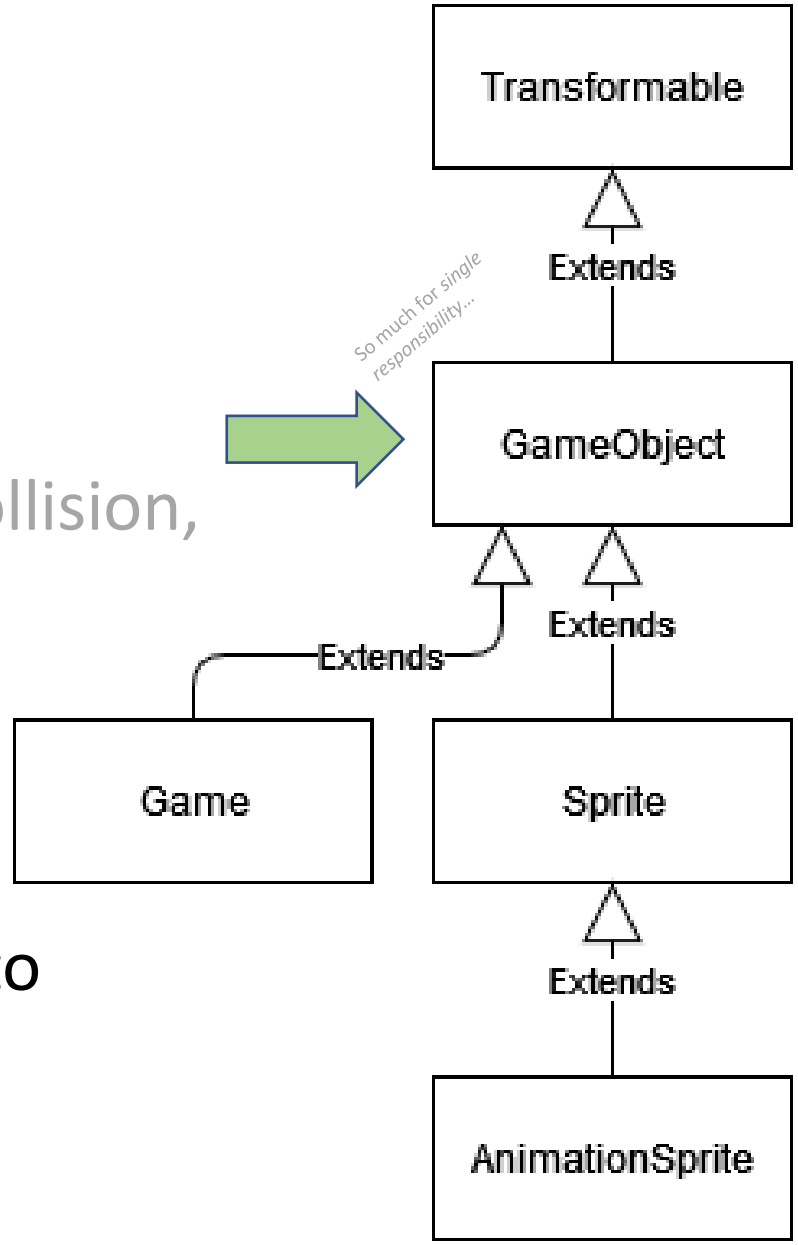
- Transformable, (Animation)Sprite, Game
- Utils.Random, Input.GetKey(Down/Up)
- …but what about *GameObject*?



# This Week: Game Objects

- Hierarchy (parent, children)
- Creating & destroying (AddChild, Destroy)
- Checking collisions (HitTest, GetCollisions, OnCollision, MoveUntilCollision)
- Interactions between different types (classes):
  - Type checking and casting

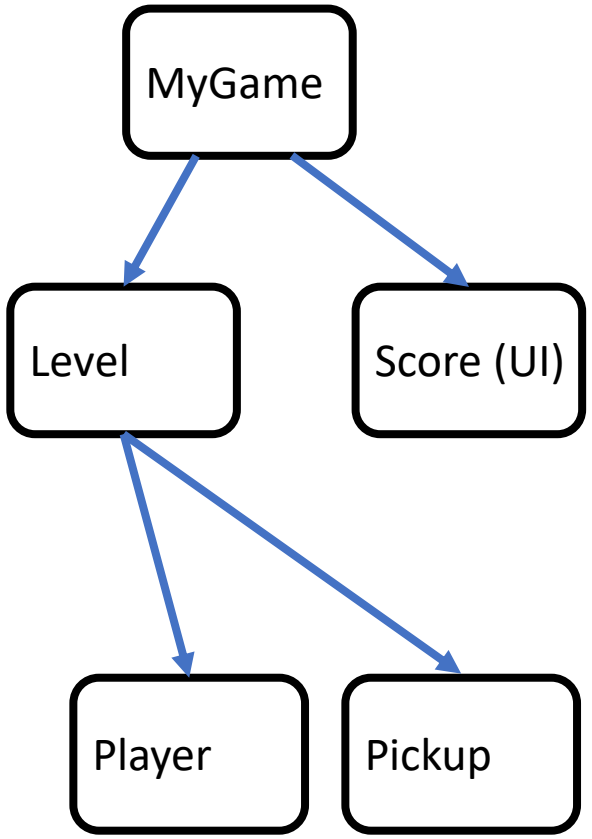
- Once you master these concepts, you're ready to *implement interesting game play*



# Poll: Progress?

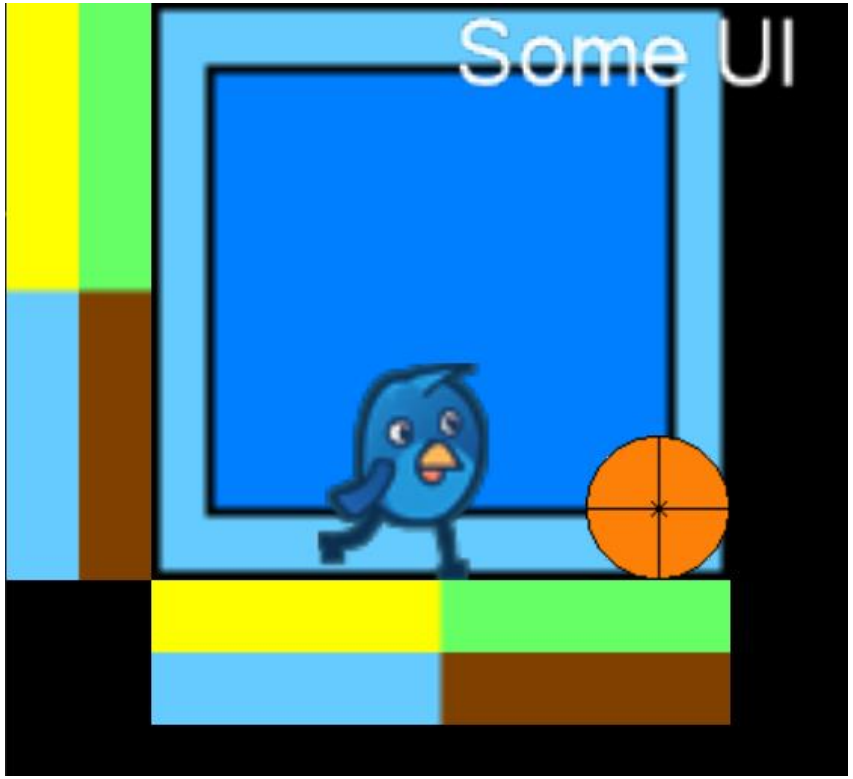
1. I basically haven’t done anything yet…
2. I have a C# IDE (like Visual Studio) installed
3. I have downloaded and run the GXPEngine
4. I have an animated, moving sprite on screen
5. I have already looked further than week 1 topics (collisions?)

# Game Object Hierarchy



# AddChild Experiments

- Let's look at a very simple example project (on Blackboard), with:
  - A Player (Barry the bird)
  - A level (represented by the blue square)
  - Two (floor / wall) tiles (the "colors" sprite)
  - A pickup (the orange circle)
  - Some UI text

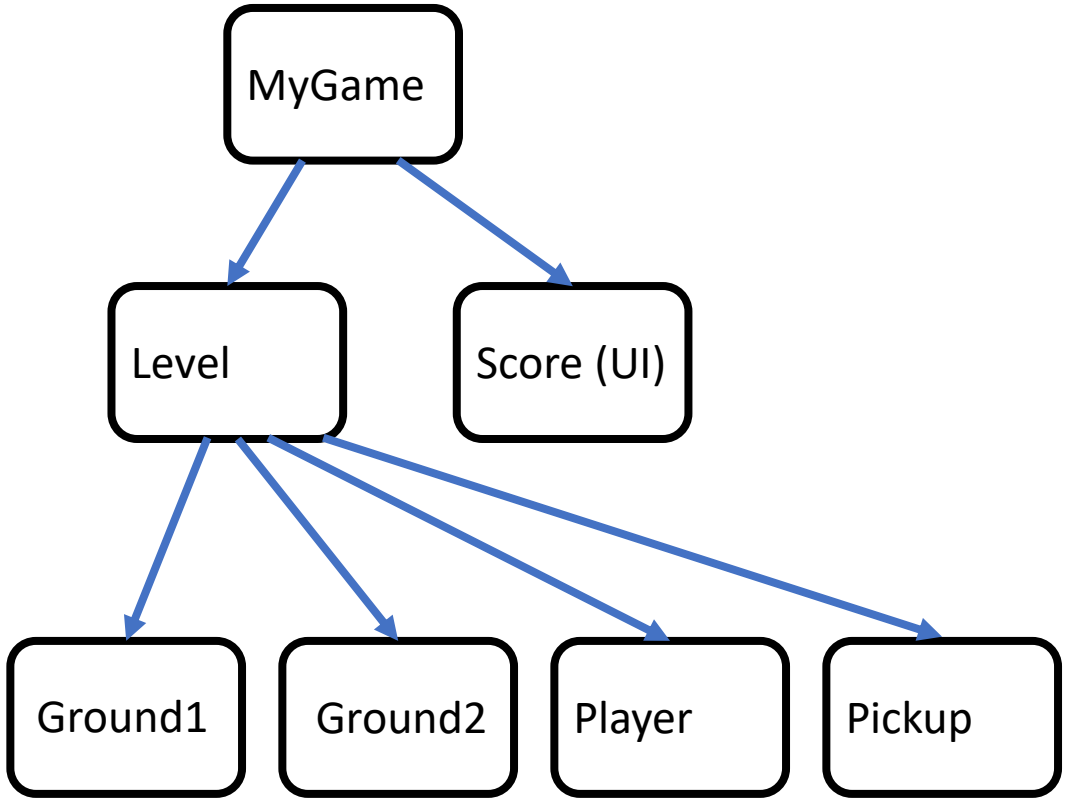


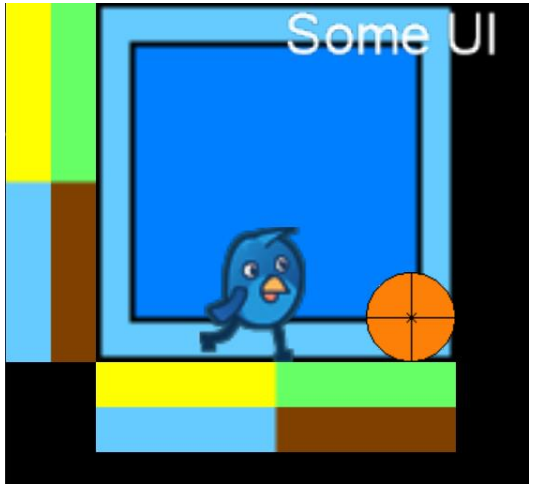

Questions:

- Can we create a *scrolling* level?
- In which *order* are the objects *rendered*?

# Hierarchy

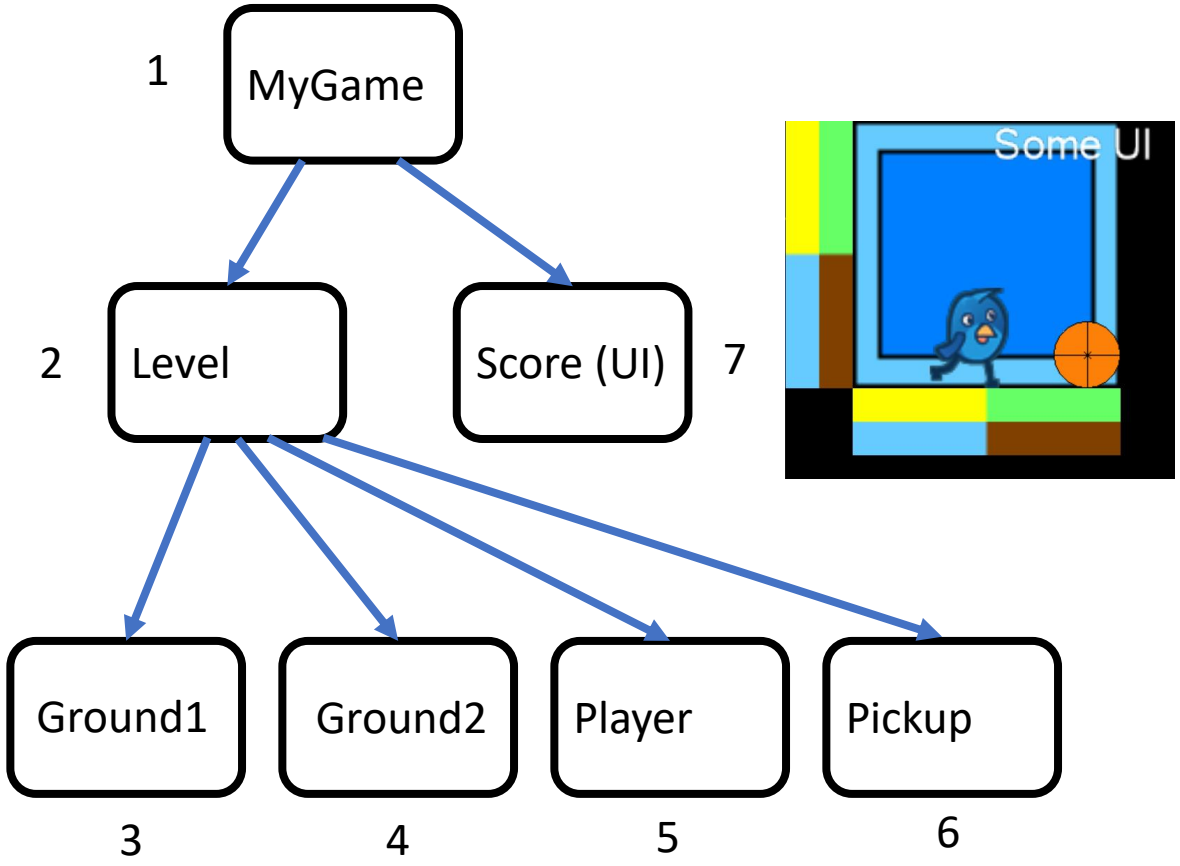
- Game objects can have any number of *children*
- Game objects have (at most) one *parent* game object
- There is one *root object* without parent: typically your *Game* (MyGame)
- In all this creates a *tree shape* (the game object *hierarchy*)
- Game objects are said to be *active* or *in the hierarchy* if they are part of the tree with Game as root



# Hierarchy

- Only *active game objects* are:
  - *Rendered* (if they are *Sprites*)
  - *Updated* (if they have an *Update* method)
  - Part of *collision checks* (if they have a *collider*)



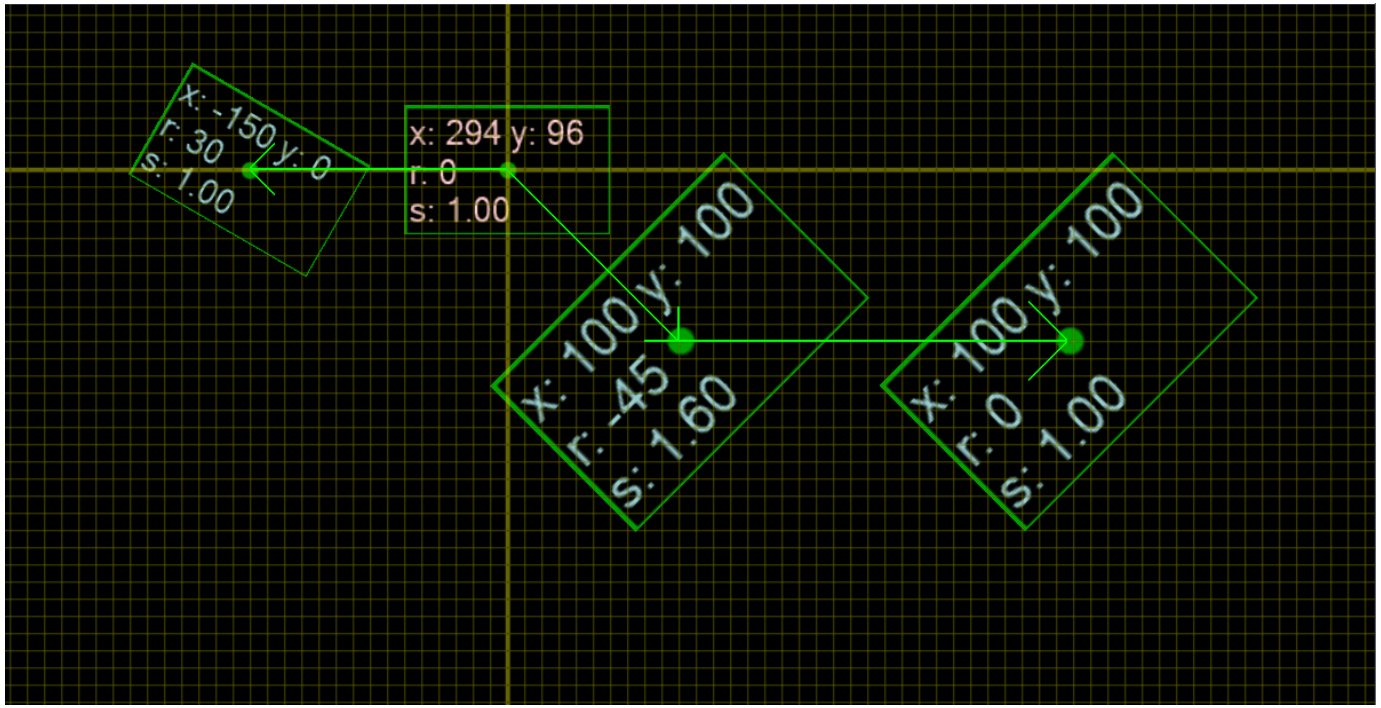

- The *render order* is determined by the child order + tree structure (can be modified – see SetChildIndex)
- The *update order* is typically determined by the order in which game objects are added to the hierarchy, but it's best to *not make assumptions* about this

# Parent / Child space

- Each game object has its *own space*, which can be rotated and scaled (rotation, scale)
- Coordinates are relative to the parent space (x,y)

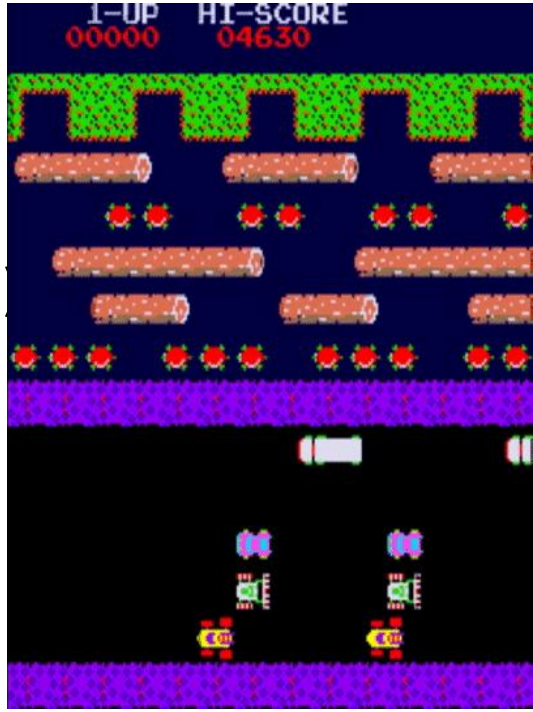
→ Let's just look at an example:

(This example is on Blackboard, under *Code Samples*)



# Why?

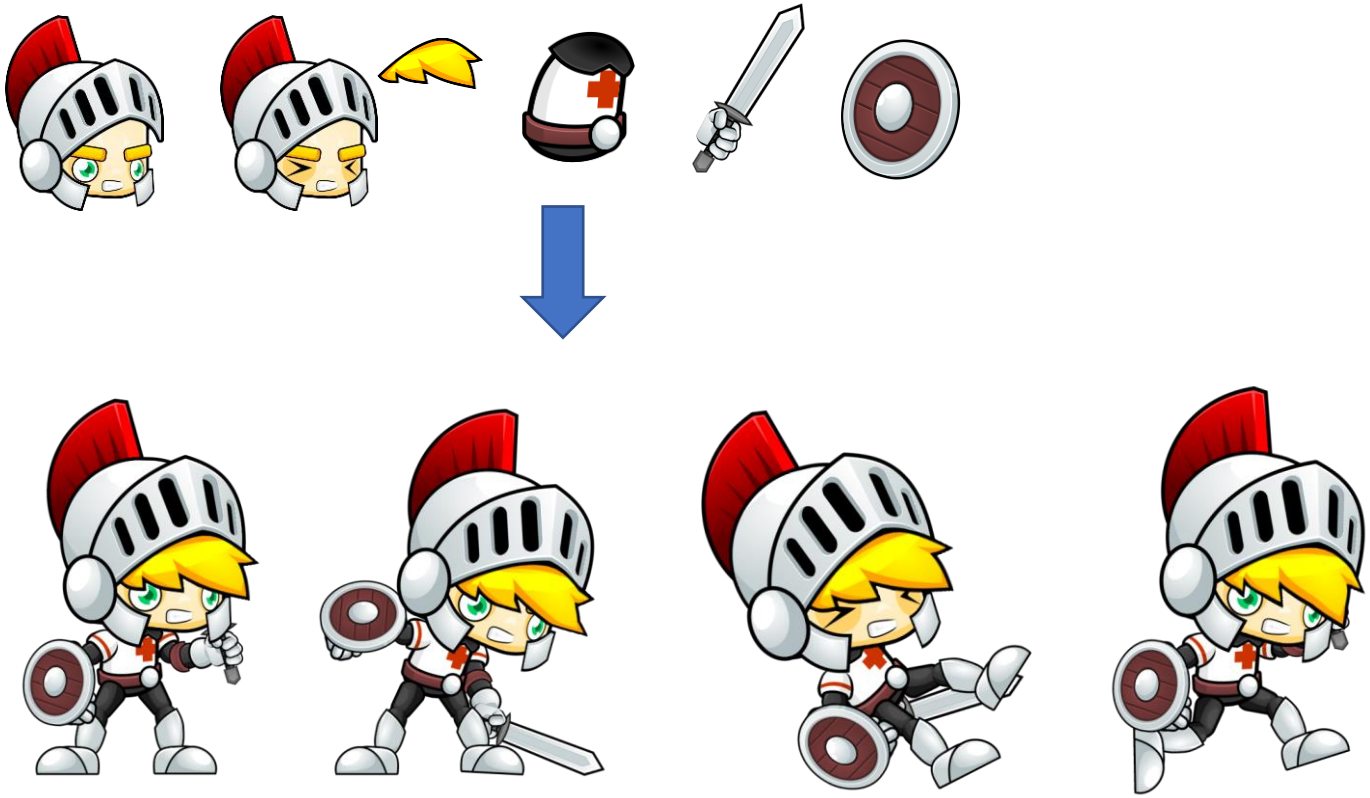
*: explained on next slides

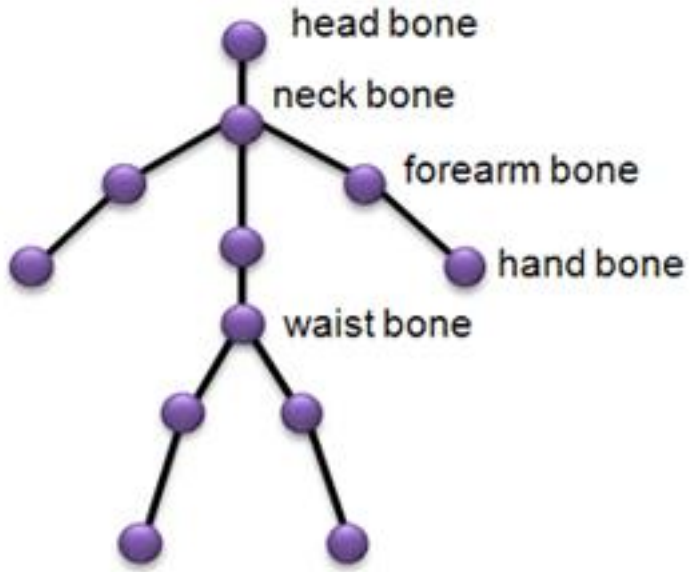



*The possibilities are endless!!*

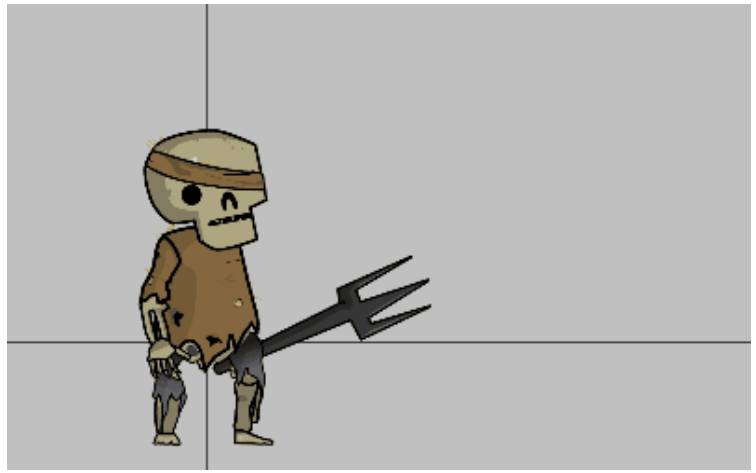
- *Items:* your character wears a hat, or carries a shield / gun / sword / …
- *Skeletal animation* (e.g. for a boss character / controlled by physics?)*
- A character stands on a *moving element* (moving platforms, Frogger logs, …
- Decouple a game object from its animations:*
  - if you have *different sprite sheets* for idle, walk, attack, with different sizes
  - If you want the 'collision box' to be smaller than the sprite
- Create a *scrolling* level, while the *UI* stays in place*
- *Parallax scrolling**
- Ensure that newly spawned objects are rendered behind the *foreground**
- Create a *rotating* level, *screen shake,* or *zooming in/out* (='camera' functionality), while all your position logic (e.g. player movement) stays simple
- You want to *create 3D visuals* using sprites: oblique, isometric → parallel projection*

# Skeletal Sprite Animation / Carrying Items



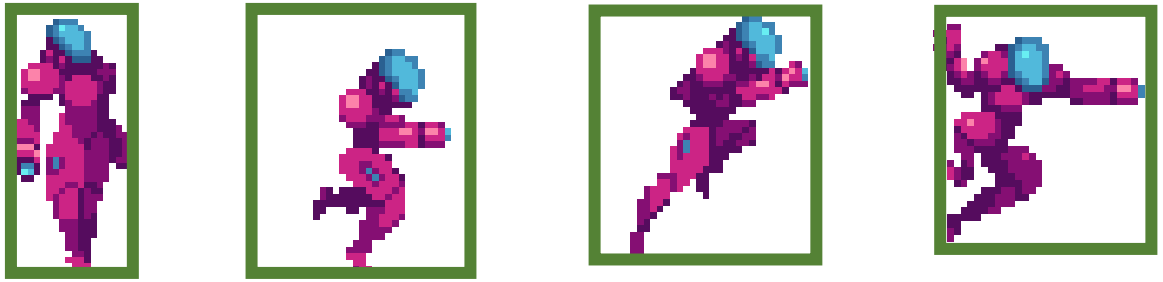



‘The Knight’ by pzUH, from opengameart.org (CC0)

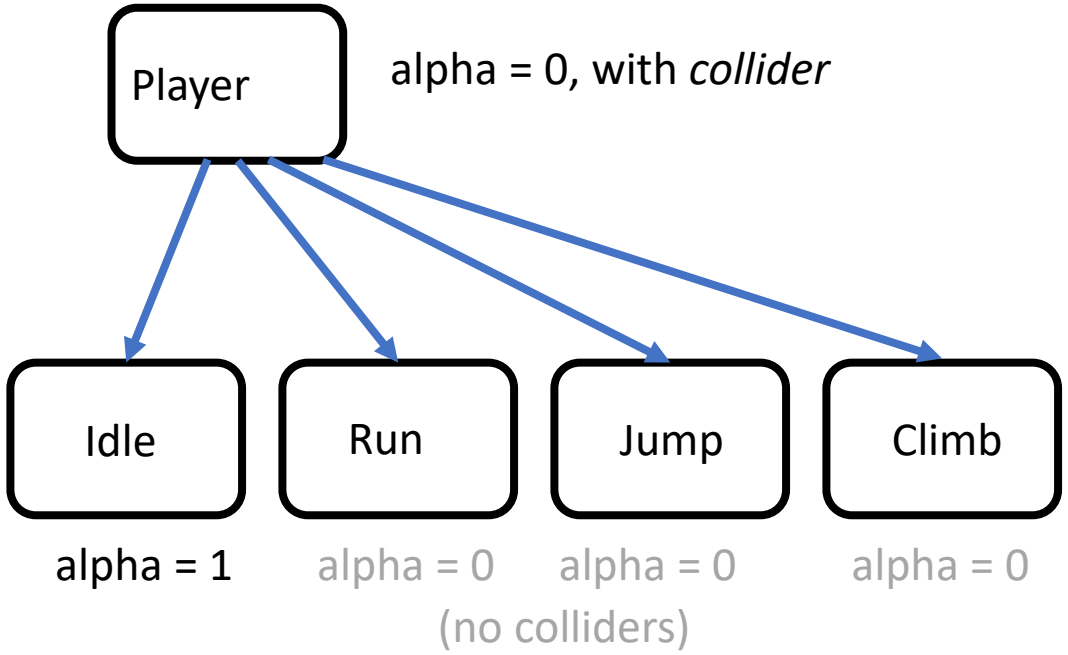


Skeleton army by Overcrafted: https://overcrafted.itch.io/skeleton-army

# Combining Different Sprite Sheets



Warped pack, by Ansimuz (itch.io / opengameart.org)



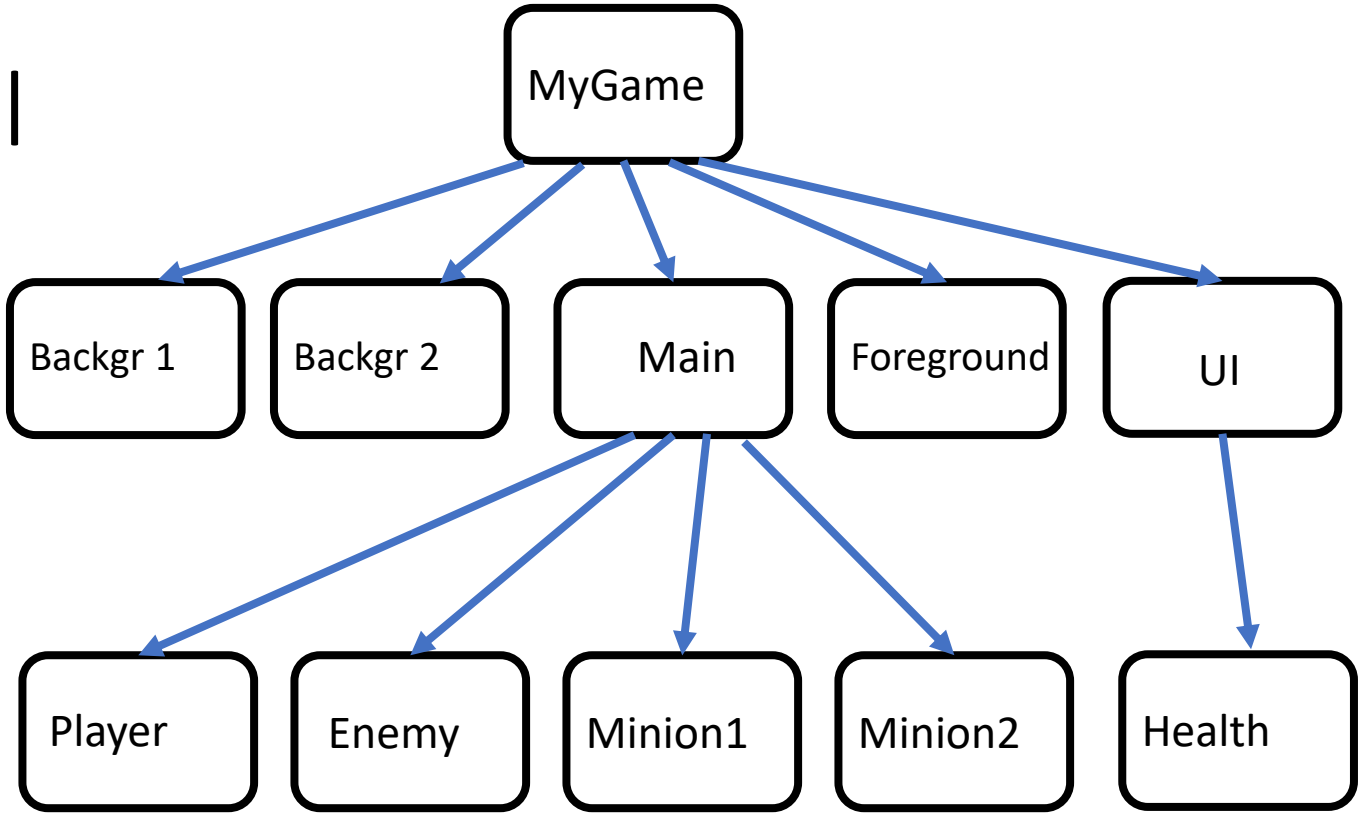

Core idea:

- Create a player game object (invisible: alpha = 0) with proper *collider size*
- Every animation cycle corresponds to an AnimationSprite child object
- Enable / disable animation cycles as desired

# Parallax Scrolling & UI

Hollow Knight (Steam - Unity)


Kenney's Epic Adventure (GXP)

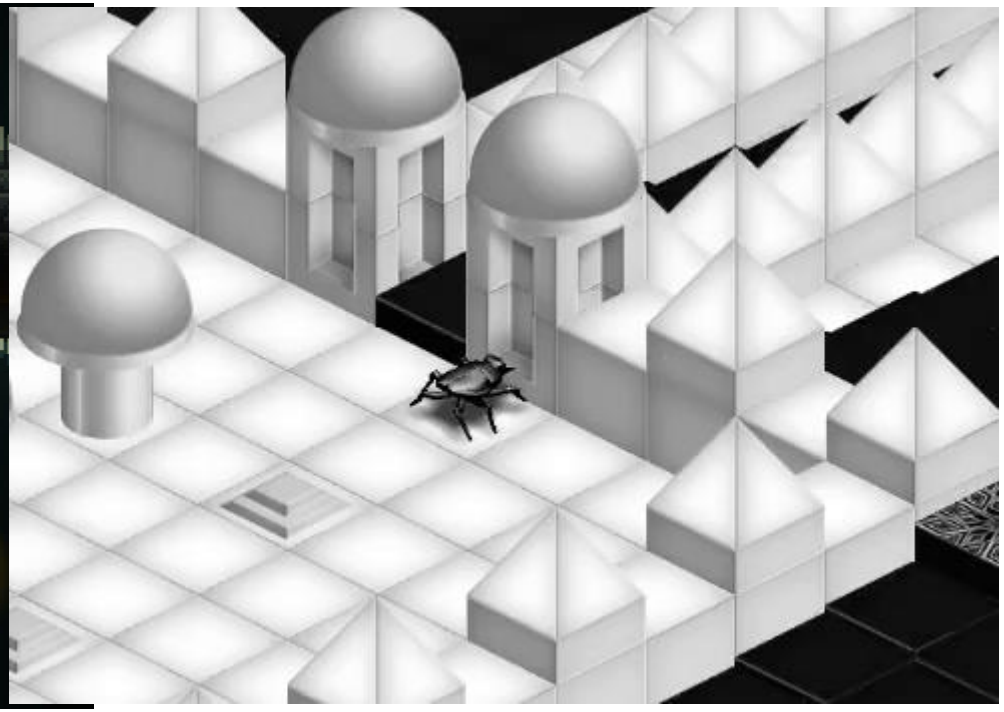



- Core idea:
  - As the player moves, all non-UI layers move in opposite direction
  - Background moves slower, foreground moves faster

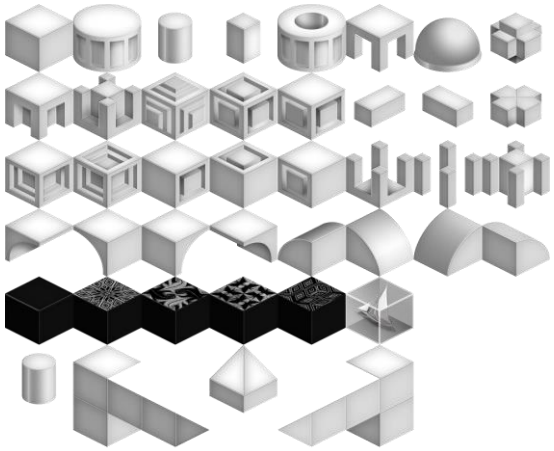
# Pseudo 3D: Parallel Projection using Sprites

See also: https://medium.com/retronator-magazine/game-developers-guide-to-graphical-projections-with-video-game-examples-part-1-introduction-aa3d051c137d

Enter the Gungeon (Steam)



Rolmops (GGJ 2017 - GXP)



Isometric sprite sheet

Core idea:

- Every *layer* is a set of sprites at the same distance from the camera (moving objects are in front or behind *all* of them)
- Layers are rendered back-to-front
- Moving elements (like the player) are moved from layer to layer (change parent) as they move vertically

I’ll assume you’re now convinced of the `why’.

But…

# How?

- Every GameObject has a *parent* GameObject (can be null – get, set)
- You can get the list of children using *GetChildren()*
- You typically use:
  - parentObject.*AddChild*(childObject)
  - Examples:
    - player.AddChild(sword)
    - mainLayer.AddChild(player)
    - AddChild(mainLayer) (adds it to the *this* object, e.g. MyGame)

# Today's Demo

- All assets by *PixelFrog* (itch.io)
- The starting code uses only techniques from last lecture / Programming Basics
- GXPE, *pixel art* settings:
  - Set *pixelArt* = true in the constructor
  - Set *realWidth* and *realHeight* as integer multiple of width and height
  - Use integer coordinates for objects


```
public MyGame() : base(320, 256, false, false, 960 , 768, true)
{
```

Game.Game(int pWidth, int pHeight, bool pFullScreen, [bool pVSync = true], [int pRealWidth = -1], [int pRealHeight = -1], [bool pPixelArt = false])
Initializes a new instance of the Game class. This class represents a game window, containing an openGL view.

# Example Code: Basic Scrolling

- To create a scrolling level, we add a Level *Pivot* (=empty GameObject), and make everything except the UI a child of it

```
void Scroll() {
    int boundarySize = 128;
    // Player is child of level, so the player's screen position = player.x + level.x
    //   (No scaling or rotation involved)
    // If the player's screen position is outside of the desired boundaries, we modify the level position:
    if (player.x + level.x < boundarySize) {
        level.x = boundarySize - player.x;
    }
    if (player.x + level.x > width - boundarySize) {
        level.x = width - boundarySize - player.x;
    }
}
```

# Dynamic Game Objects

- You can create new game objects during the game, e.g. from Update
  - Example: a player or enemy shoots a bullet
  - To prevent confusion, it may be good to use *LateAddChild:* adds the game object after all *Updates* and *OnCollisions* have been called by the engine
  - *Note:* It's a bad idea to change a list while you (or actually the engine) are looping through it!
- Calling *Destroy()* will destroy a game object, and all of its children
  - Example: a bullet hits a wall
  - Similar to above, using *LateDestroy()* might be better
- If you want to remove an object *temporarily* from the hierarchy, you can also call *Remove()* or simply set *parent=null*

# Demo: shooting projectiles

Note:

- When creating a Bullet from Player.Update(), you typically want to do parent.AddChild(), not AddChild!
- Don’t forget to Destroy projectiles that are out of screen!
- Shoot cooldown:
  - *Time.time* gives the current time in milliseconds

```
if (Time.time>lastShootTime + shootIntervalMs) {
    StartShoot();
    lastShootTime = Time.time;
}
```

# Collisions

# Object Interaction and Game Play

- *Game play* comes from *interaction between game objects*
- Examples:
  - Enemy hits player (decrease player health)
  - Player grabs pickup (increase ammo / health)
  - Player bullet hits enemy (destroy enemy)
  - Player reaches end of level gate (load next level)
- Game objects interact when they *overlap*, or *collide*
- What we need now:
  - Check for overlapping objects (or *collisions*)
  - Determine the *type* (class) of the object, and react appropriately

At least, this holds for typical arcade games, platformers, action games, shooters, etc.

# Checking for Collisions (Overlaps)

- *HitTest(other)* returns true if *this* game object overlaps with some *other* game object
- *GetCollisions()* returns an array of all overlapping game objects

# Demo

- Player grabs pickup (HitTest)
- Bullets are destroyed when hitting anything (GetCollisions)

- *Note:* in a moment we'll see how we can do this *with* background sprites

# Code Quality - 2

We want our code to be:

- *Reusable:* ideally you can easily reuse some of your classes in the upcoming projects
- *Maintainable:* If you look at your code weeks from now to fix a bug, it should be clear where and how to do it - ideally with few changes
- *Extensible:* Your code should allow for adding new features in the future
- *Robust:* It should be hard to create bugs (also for other team members!)

- Many of the code *guidelines* we have seen / will see serve these purposes (DRY, low coupling, encapsulation, short methods, avoid global or class level variables, etc.)
- There are always different ways to set up your code. *Make a choice with these things in mind, and be ready to explain it.*

# Checking for Collisions (Overlaps)

- *HitTest(other)* returns true if *this* game object overlaps with some *other* game object
  - *Advantage*: *efficient* overlap check
  - *Disadvantage*: you need to have a reference to the other game object
    - Example: Player maintains a list of all Pickups in the level, or Pickups have ref. to Player
    - *Bad code!* Unmaintainable, inflexible, error prone! (What if a pickup was already destroyed? What if you decide to implement local multiplayer?)
- *GetCollisions()* returns an array of all overlapping game objects
  - *Advantage*: clean code – minimal *coupling* between classes
  - *Disadvantage*: this gets *inefficient* and slow when there are many game objects with colliders!

# (Box)Colliders

- The GXPEngine only contains *BoxColliders*, which are rectangle shaped, exactly as large as the sprite
  - (During Physics Programming, we'll see how to implement circle colliders, line segments, polygons, etc.)
- *Sprites* (and classes that inherit from it) have BoxColliders by default
- Set *addCollider=false* in the constructor to *omit* the collider
- Game objects without colliders are ignored by *all* collision check methods
- This is useful for background / foreground tiles:
  - just decoration, no interaction
  - Makes GetCollisions a lot more efficient!

These two guys currently collide

# OnCollision

- If you add a method `void OnCollision(GameObject other)` to your class, it will be called automatically each frame, for each overlapping game object other (with collider)
  - *Advantage*: maybe even more convenient than GetCollisions
  - *Disadvantage*: maybe even more inefficient than GetCollisions

- Demo (Bullet)
- *Note:* LateDestroy needed!

# Type Checking

- Player collisions:
  - Pickup: increase health / ammo
  - Bullet/Enemy: decrease health
  - Wall: stop moving

- How can we implement this?

- *Type checking: is* keyword (C#)

```
if (other is Pickup) { … }
```

# Casting

- For every GameObject other, we can call the Destroy or SetXY methods. Doesn't matter whether it's a Pickup, Wall, Bullet, …
- However, even when we know it's a Bullet, we cannot call the GetDamage() method…?
  - This requires (explicit) *casting*. C# provides two methods:
    - `Bullet bullet = (Bullet)other;` (may give Exception)
    - `Bullet bullet = other as Bullet;` (may return *null*)

# Player Collision Code

```
float oldX = x;
float oldY = y;

x += dx;
y += dy;

// handle collisions:
GameObject[] collisions = GetCollisions();
for (int i=0;i<collisions.Length;i++) {
    if (collisions[i] is Enemy || collisions[i] is Shooter) {
        TakeDamage(1);
        collisions[i].Destroy(); // No casting needed
    } else if (collisions[i] is Pickup) {
        ((Pickup)collisions[i]).Grab(); // Casting needed
        GrabAmmo();
    } else if (collisions[i] is Solid) {
        // move back:
        x = oldX;
        y = oldY;
    }
}
```

# Enemy Collision

- Next, we’ll make our flying enemy turn around when it hits a wall

- Demo

# Enemy Collision – Stuck in Wall Bug

```
21  void Update() {
22      float oldX = x;
23
24      // starting point:
25      x += vx;
26
27      if (x<0 || x > game.width / game.scaleX) {
28          TurnAround();
29      }
30      // collision checking:
31      GameObject[] collisions = GetCollisions();
32      for (int i=0;i<collisions.Length;i++) {
33          if (collisions[i] is Solid) {
34              TurnAround();
35              x = oldX;
36          }
37      }
```


Looks okay though…?

# Debugging Tools

- In many cases, Console.WriteLine is sufficient for debugging, but you can do more:

- You can set *breakpoints* to *pause* the code when a certain line is reached (Visual Studio: F9)
- When paused, you can inspect:
  - Values of *local variables*
  - The *call stack*
- If you don't see them in Visual Studio *while paused*, go to Debug →Window

# Finding the bug

```
27      if (x<0 || x > game.width / game.scaleX) {
28          TurnAround();
29      }
30      // collision checking:
31      GameObject[] collisions = GetCollisions();
32      for (int i=0;i<collisions.Length;i++) {
33          if (collisions[i] is Solid) {
34              TurnAround();
35              x = oldX;
36          }
```

141 % | No issues found | Ln: 34 Ch: 5 Col: 17 TABS CRLF

Locals

Search (Ctrl+E) | Search Depth: 3

| Name | Value | Type |
|---|---|---|
| this | {[Enemy::Bird.png]} | Enemy |
| oldX | 32 | float |
| collisions | {GXPEngine.GameObject[2]} | GXPEngine.GameObject[] |
| [0] | {[Solid::Terrain (16x16).png]} | GXPEngine.GameObject... |
| [1] | {[Solid::Terrain (16x16).png]} | GXPEngine.GameObject... |
| i | 0 | int |

Call Stack

| Name |
|---|
| GXPEngine.exe!Enemy.Update() Line 34 |
| GXPEngine.exe!GXPEngine.Managers.UpdateManager.Step() Line 27 |
| GXPEngine.exe!GXPEngine.Game.Step() Line 186 |
| GXPEngine.exe!GXPEngine.Core.GLContext.Run() Line 221 |
| GXPEngine.exe!GXPEngine.Game.Start() Line 176 |
| GXPEngine.exe!MyGame.Main() Line 134 |

Aha! We're turning around twice!

# Fixing the Bug

```
// collision checking:
GameObject[] collisions = GetCollisions();
for (int i=0;i<collisions.Length;i++) {
    if (collisions[i] is Solid) {
        TurnAround();
        x = oldX;
        break;
    }
}
```

# Collisions with Solid Objects

- When a moving object collides with a solid object (e.g. a Player hits a Wall), we want it to *stop moving / resolve the collision* (no overlap).
- *Easy recipe:*
  - Store the old position (x,y)
  - Try to move
  - Check for collisions
  - If there's a collision with a solid object: move back to the old position
- In many cases this is good enough, but…
- *Problem:* the object might stay stuck, a few pixels away from the wall or ground!

# MoveUntilCollision

- *Problem:* the object might stay stuck, a few pixels away from the wall or ground!
- *Solution:* *MoveUntilCollision(dx, dy)* tries to move the game object by dx, dy (same as Translate(dx,dy)), but stops when it hits a (solid) collider

- *Pro tip:* separating x and y movement can prevent getting stuck in walls / on the ground when moving diagonally:

```
// With getting stuck on wall/ground:
MoveUntilCollision(dx, dy);

// Without getting stuck on wall/ground:
MoveUntilCollision(dx, 0);
MoveUntilCollision(0, dy);
```

# Triggers

- *Problem:* The Player can now also not move through Pickups…
- *Solution:* Make the Pickup a *trigger*
- In the Pickup class: `collider.isTrigger = true;`
- Note that MoveUntilCollision still works for game objects with trigger colliders (yay!)

- (Demo)

# Configuring your Collisions - Tips

- Move your objects using the “easy recipe” *(solid collision → move back to old position)*, or using *MoveUntilCollision*
- When using MoveUntilCollision: Make objects like Player, Enemy, Pickup, Bullet into *triggers*
- Solid objects like walls, ground should never be triggers
- Check for interactions using *GetCollisions*, or *OnCollision*

- If you want to improve *efficiency*, you can use *HitTest*, or pass a specific list of objects to *MoveUntilCollision*
  - More details later!

# Code Quality - Refactoring

- Our Player.Move method is now quite long…
- How can we improve this?
- …without *introducing unnecessary class-level variables?!*

- *Answer:* methods with *parameters* and *return values* (demo)

```
void Move() {
    float dx = 0;
    float dy = 0;
    if (Input.GetKey(Key.LEFT)) {
        dx = -speed;
        Mirror(true, false);
    }
    if (Input.GetKey(Key.RIGHT)) {
        dx += speed;
        Mirror(false, false);
    }
    if (Input.GetKey(Key.UP)) {
        dy = -speed;
    }
    if (Input.GetKey(Key.DOWN)) {
        dy += speed;
    }

    float oldX = x;
    float oldY = y;

    x += dx;
    y += dy;

    // handle collisions:
    GameObject[] collisions = GetCollisions();
    for (int i=0;i<collisions.Length;i++) {
        if (collisions[i] is Enemy || collisions[i] is Shooter) {
            TakeDamage(1);
            collisions[i].Destroy(); // No casting needed
        } else if (collisions[i] is Pickup) {
            ((Pickup)collisions[i]).Grab(); // Casting needed
            GrabAmmo();
        } else if (collisions[i] is Solid) {
            // move back:
            x = oldX;
            y = oldY;
        }
    }

    if (dx!=0 || dy!=0) {
        SetCycle(11, 12); // run
    } else {
        SetCycle(0, 11); // idle
    }
}
```

# Summary

- Game object *hierarchy*, and all the nice things you can do with it (like scrolling)
- *Adding* and *destroying* game objects
- *Collision checking* (HitTest, GetCollisions, OnCollision, MoveUntilCollision)
- *Type checking* and *casting*
- *Debugging* (break points, locals, call stack)
- Other small things: *Pivot, pixel art, Time.time*
- More on code quality:
  - Things we want: *reusable, maintainable, robust, extensible*
  - Some ways to achieve it: *low coupling, few class level variables, short methods*

# During the Lab

- Create *different* (moving*) object types* and *interactions* between them (player, enemy, pickup, projectile, wall?)
- Use *type checking* and *casting* + some of the *collision checking* methods
- Keep your code clean! (maintainable, extensible, reusable, robust) → *ask for feedback*

*Optional:*

- Experiment with some of the mentioned `hierarchy tricks' (scrolling, screen shake, carrying items, …?)

- These are little (or big) programming exercises: if you're stuck, ask for help!
- Remember: *if it's easy, you're not learning ;-)*

# Next Week

- Using these concepts, you can already *create interesting game play*
- However, …
  - Game object positions are hard coded or random…
  - You only have one level…
  - Game objects only interact when they overlap…

Next week:

- Creating and loading levels using *arrays* or *Tiled*
- Switching levels (and keeping track of information)
- Linking objects (references)