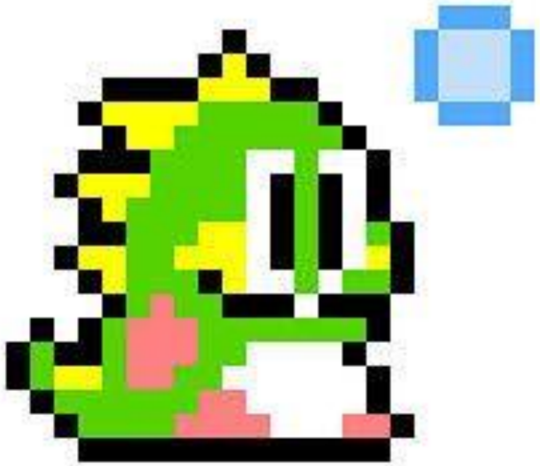
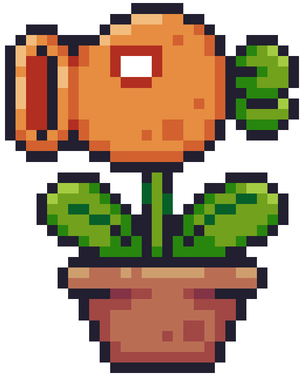
Game Programming

Lecture 6: Advanced topics / project & assessment preparation

January 23, 2023

Paul Bonsma





Status?

- Code reviews done?
- Playable game?
- Complete game? (Start screen, HUD, sounds, etc.)
- Is it fun?
- Problems with scale/efficiency?

Struggling?

- *Go to the labs and get feedback / tips!*
- Check out the basic *video tutorials* on Blackboard
- Make sure you got the *(Programming) Basics* covered
- *Scope down!*
  - NO: Super Mario, Legend of Zelda: LTTP, Super Meat Boy, Celeste, …
  - YES: Frogger, Tapper, Breakout, Space Invaders, Flappy Bird, Doodle Jump, …

This Lecture

- Advanced Topics:
    - Preparation for upcoming project(s)
    - Enabling you to create your dream game
    - Not all necessary to pass this course though
- Assessment preparation

Outline

- Sharing your work
- Optimization
- Extra features
    - Cameras
    - Working with (text) files
- Assessment preparation

Sharing your Work

Sharing Your Work

Why?

- Share “build”:
  - Play testing (for Game Design?)
  - Show to friends and family
  - Portfolio
  - You made an awesome game and people deserve to play it! (itch.io?)
- Share code / project:
  - Code reviews (see grading criteria)
  - Get graded on Blackboard
  - Share with team members (upcoming projects)

Build

- You have a build as soon as you press “play” in your IDE(!)
- Just check bin/Debug: find the .exe file. Double click it.
- Distributing your build: zip the entire contents of bin/Debug (including assets, lib), send it to someone

Possibly:

- Remove unused assets
- Remove the .pbd file (debug info)

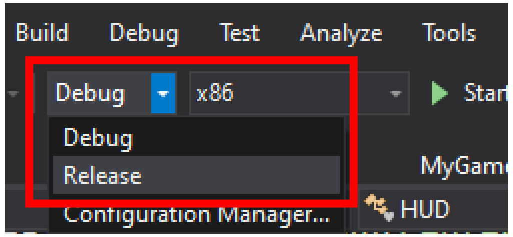
Pitfalls

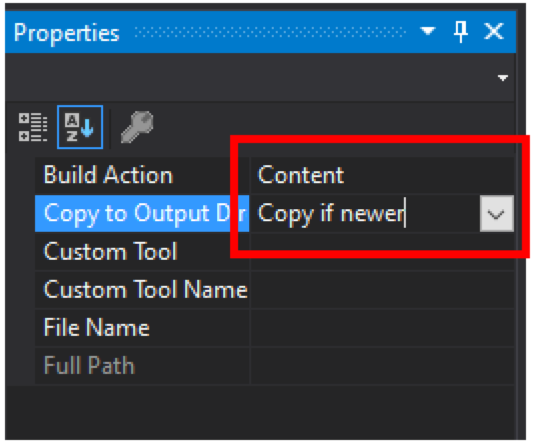
- If you use assets outside of bin/Debug, this won’t work!
- Most common case: you messed up in your Tiled map, and imported tile sets from outside bin/Debug…
- Fix: Open your Tiled map in a text editor (notepad++) and change the references to outside files (typically those that start with “../..”)

```
<tileset firstgid="418" name="Idle (32x32)" tilewidth="32" tileheight="32" tilecount="11" columns="11">
 <image source="../../../../../../../Assets/Sprites/PixelFrog/PixelAdventure_by_PixelFrog/Pixel Adventure 1/Main
 Characters/Virtual Guy/Idle (32x32).png" width="352" height="32"/>
</tileset>
```

```
<tileset firstgid="418" name="Idle (32x32)" tilewidth="32" tileheight="32" tilecount="11" columns="11">
 <image source="Idle (32x32).png" width="352" height="32"/>
</tileset>
```

Release Build





Build Action: Content

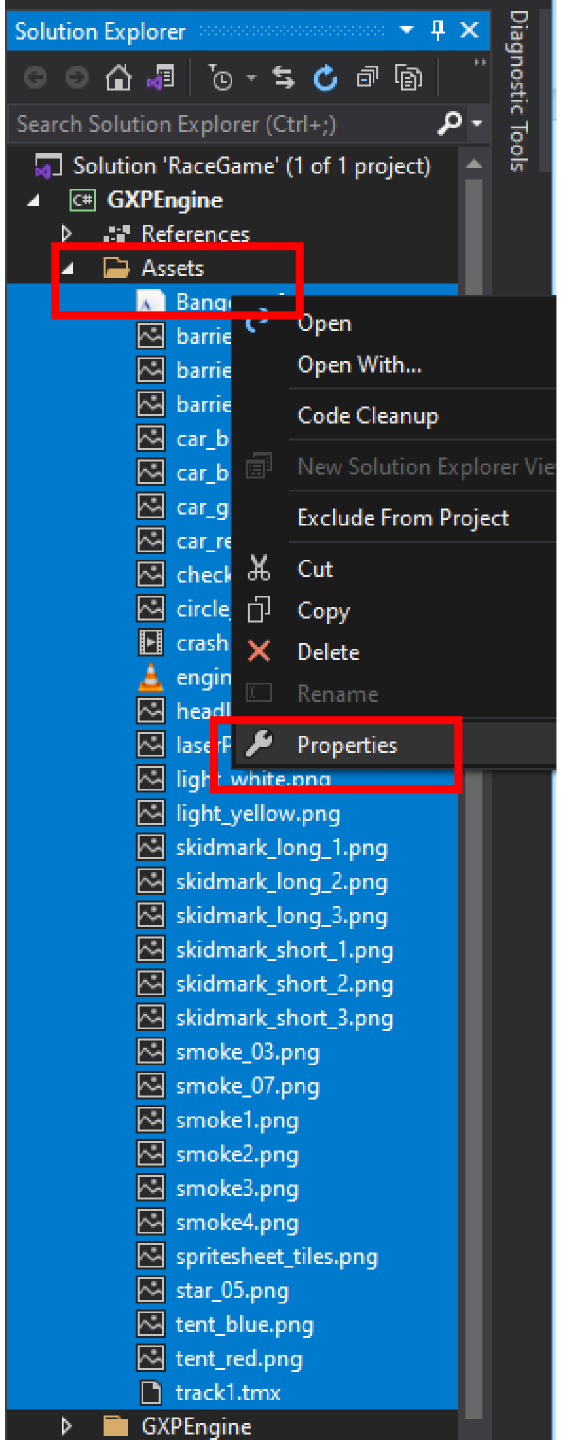
Copy to Output Dir: Copy if newer

Custom Tool

Custom Tool Name

File Name

Full Path



More professional way:

- Create a Release build (change “Debug” to “Release” in your IDE)
- → Runtime exception! Why?
  - Copy *lib* folder from bin/Debug to *bin/Release*
  - Copy *assets* from bin/Debug to *bin/Release*
- Even more professional:
  - add the used assets to a new *content folder* in Visual Studio
  - Select them, right-click → properties: create a *copy build action*

Problems

- Many tools (Teams, Google Drive) don’t allow you to distribute zip files that contain executables…
- Some virus scanners even give (false) malware warnings!

→Another reason to share the whole project instead

→…without the executables!

Clean Solution

- Remove the *.exe* and *.pdb* files from bin/Debug and bin/Release
- Remove the entire *obj folder* (that's GXPEngine/obj)

- If you're lucky, *Build* → *Clean Solution* might work too

- Then you're ready for a Blackboard upload

- Note: knowing what the temporary files are is also relevant when working with version control (git): *never commit temporary files!* (.gitignore)

Optimization

Frame Rate

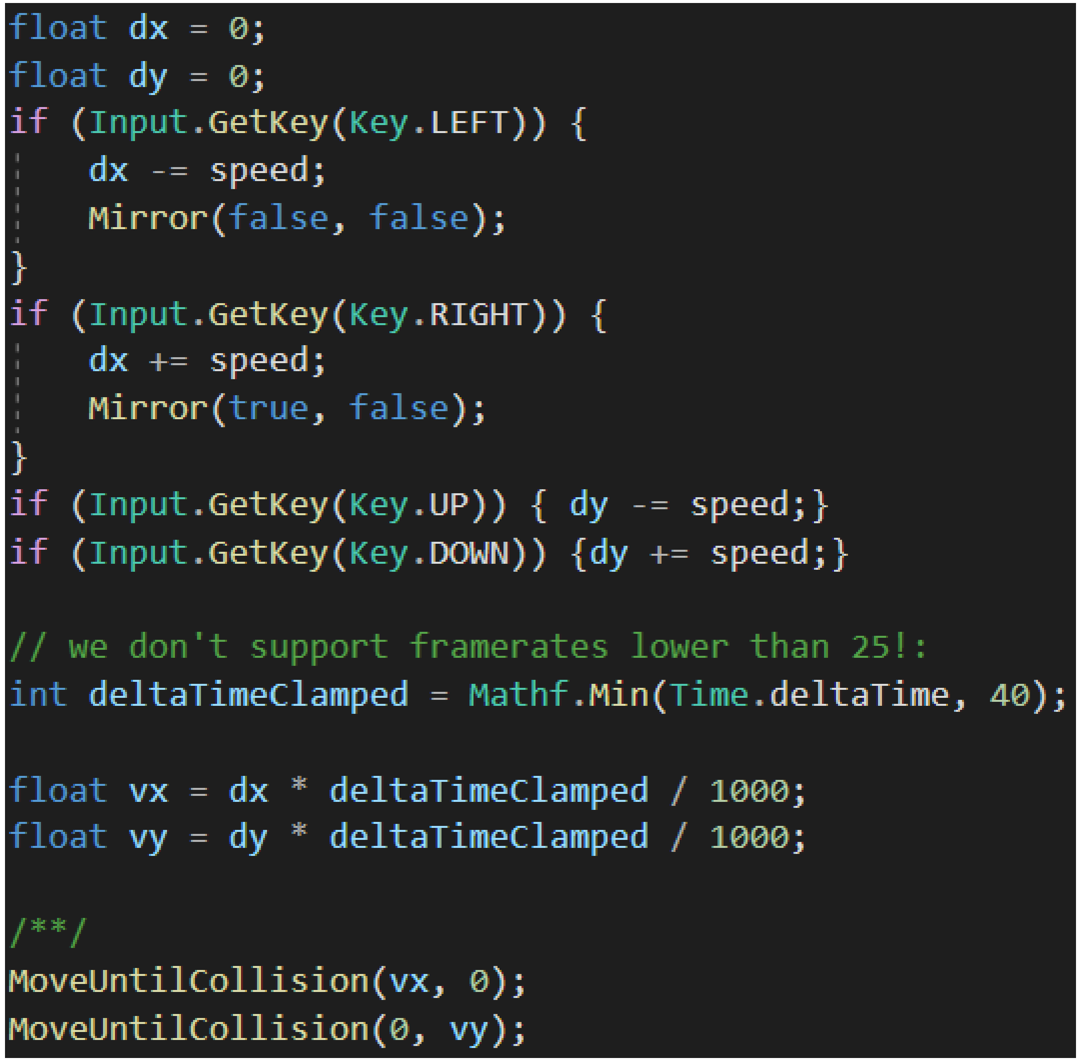
- You can (should?) guarantee that your project runs at 60Hz everywhere (even on a potato):
  - Program efficient code
  - Use VSync (sync with refresh rate) on 60Hz monitors
- In MyGame, check out the methods / properties:
  - *SetVSync, targetFps, currentFps*

- Vsync, high refresh rate monitors → too fast movement
- Run on potato and/or bad programming → too slow movement

  → *Time.deltaTime* to the rescue!

Delta Time

- If you want to deal with different frame rates:
  - use *Time.deltaTime* for all your movement
- Time since last frame (in ms)
- At 60Hz, Time.deltaTime is ~16
- *(demo)*
- *Tip:* While resizing/dragging the window, it freezes, so you'll get one huge deltaTime value! → *clamp* your deltaTime values to prevent weird movement / collision issues! (max: 40?)



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

// we don't support framerates lower than 25!:
int deltaTimeClamped = Mathf.Min(Time.deltaTime, 40);

float vx = dx * deltaTimeClamped / 1000;
float vy = dy * deltaTimeClamped / 1000;

/**/
MoveUntilCollision(vx, 0);
MoveUntilCollision(0, vy);
```

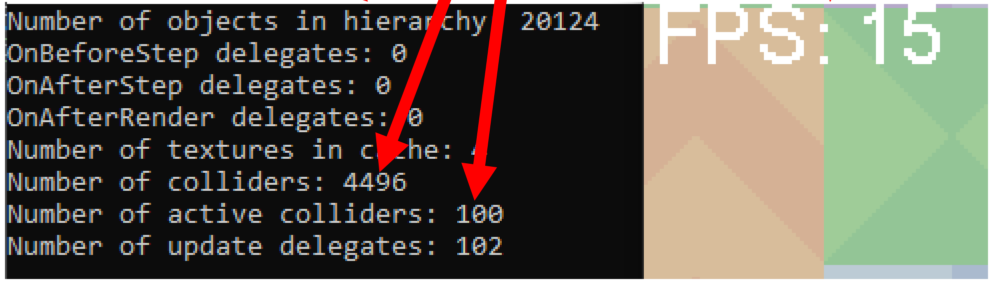
Bad Performance - Reasons

- When using large (scrolling) levels, there are two common reasons for bad performance (=low frame rate):
  - *Rendering* many sprites (“many”= more than 10000, depending on machine)
  - *Checking collisions* for many objects.
    - These methods take time proportional to the *total number of colliders:*
      - GetCollisions
      - OnCollision
      - MoveUntilCollision (all *non-trigger* colliders)
- Use *GetDiagnostics()* to get to the bottom of it!

>20.000 sprite draw calls

100 * 4495 > 440.000 collision checks!

Result: unhappy computer

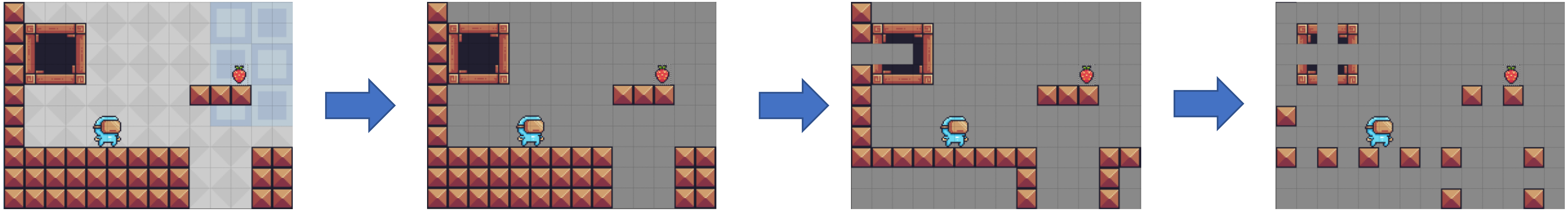


```
Number of objects in hierarchy: 20124
OnBeforeStep delegates: 0
OnAfterStep delegates: 0
OnAfterRender delegates: 0
Number of textures in cache: 4
Number of colliders: 4496
Number of active colliders: 100
Number of update delegates: 102
```

Number of objects with an *OnCollision* method

Bad Performance – Simple Fixes

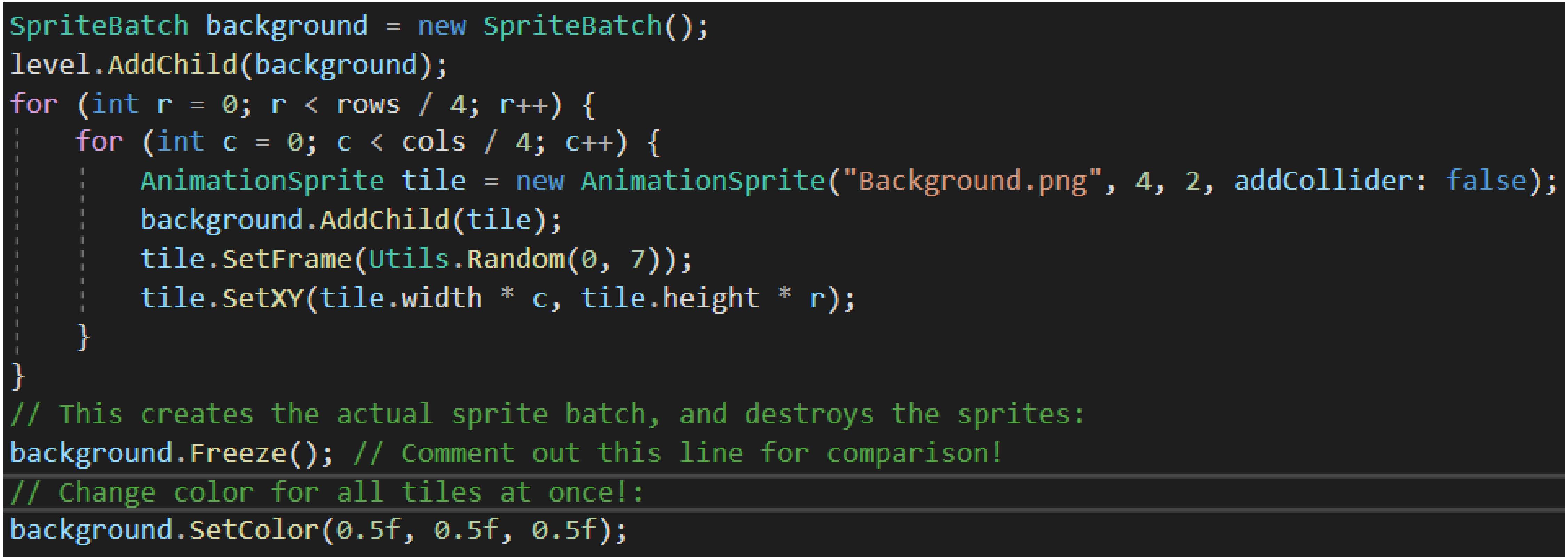
- Don’t use large levels… (Digger, Lode Runner, Pacman, Breakout, Tetris, Bubble Bobble → all awesome on a single screen!)
- Don’t use colliders unless needed! (addCollider=false for particles, HUD, background tiles)
- …don’t use colliders unless *really* needed! (Not even all foreground tiles require colliders!)
- Use *trigger* colliders when possible (makes MoveUntilCollision faster)



Rendering Optimization: SpriteBatch

- You can use a *SpriteBatch* to render many sprites in one draw call
- Disadvantages:
  - These sprites cannot move / rotate / change color /… individually anymore
  - These sprites don’t have collisions (colliders) anymore
- Advantage:
  - You can change the color / position / … of all of these sprites with one command!
- Conclusion:
  - Typical use: background tiles that don’t change, or foreground tiles without colliders

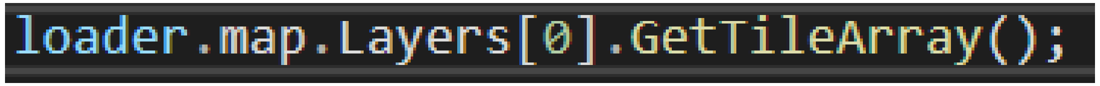
SpriteBatch



```
SpriteBatch background = new SpriteBatch();
level.AddChild(background);
for (int r = 0; r < rows / 4; r++) {
    for (int c = 0; c < cols / 4; c++) {
        AnimationSprite tile = new AnimationSprite("Background.png", 4, 2, addCollider: false);
        background.AddChild(tile);
        tile.SetFrame(Utils.Random(0, 7));
        tile.SetXY(tile.width * c, tile.height * r);
    }
}
// This creates the actual sprite batch, and destroys the sprites:
background.Freeze(); // Comment out this line for comparison!
// Change color for all tiles at once!:
background.SetColor(0.5f, 0.5f, 0.5f);
```

Rendering Optimization: Advanced

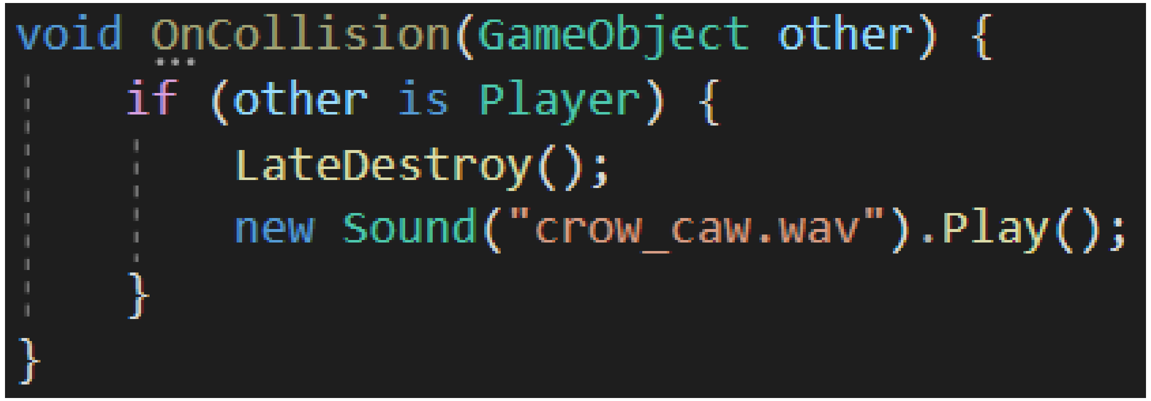
- You only need to render sprites that are actually on screen…
- Scrolling trick:
  - Store tile numbers (=animation frames) in a *2D array*
  - Create just enough animation sprites to cover the screen
  - Whenever a tile leaves the screen on the left, make it reappear on the right, and change the frame! (Similar for top → bottom, etc.)
- This also works when you want to destroy / create tiles dynamically, or change their color individually
- *Demo*
- When using a TiledLoader, you can get the array using e.g.:



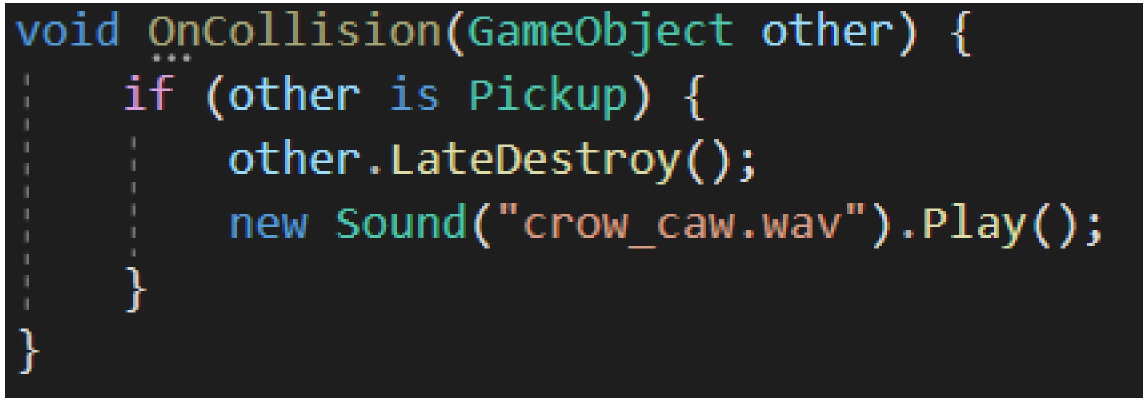
```
loader.map.Layers[0].GetTileArray();
```

Collision Optimization 1

- Simple collision optimization:
- Say you have 1 player, and 100 pickups in your level (=101 colliders)
- Should you do OnCollision (or GetCollisions) in your Player class or in your Pickup class?
- In Player class! (If in Pickup class: 100x as many collision checks!)



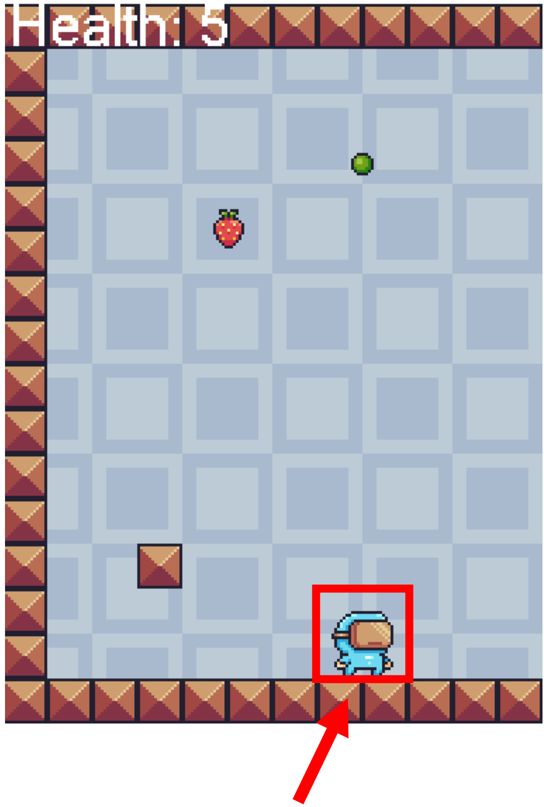
```
void OnCollision(GameObject other) {
    if (other is Player) {
        LateDestroy();
        new Sound("crow_caw.wav").Play();
    }
}
```



```
void OnCollision(GameObject other) {
    if (other is Pickup) {
        other.LateDestroy();
        new Sound("crow_caw.wav").Play();
    }
}
```

Collision Optimization 2: Space Partitioning

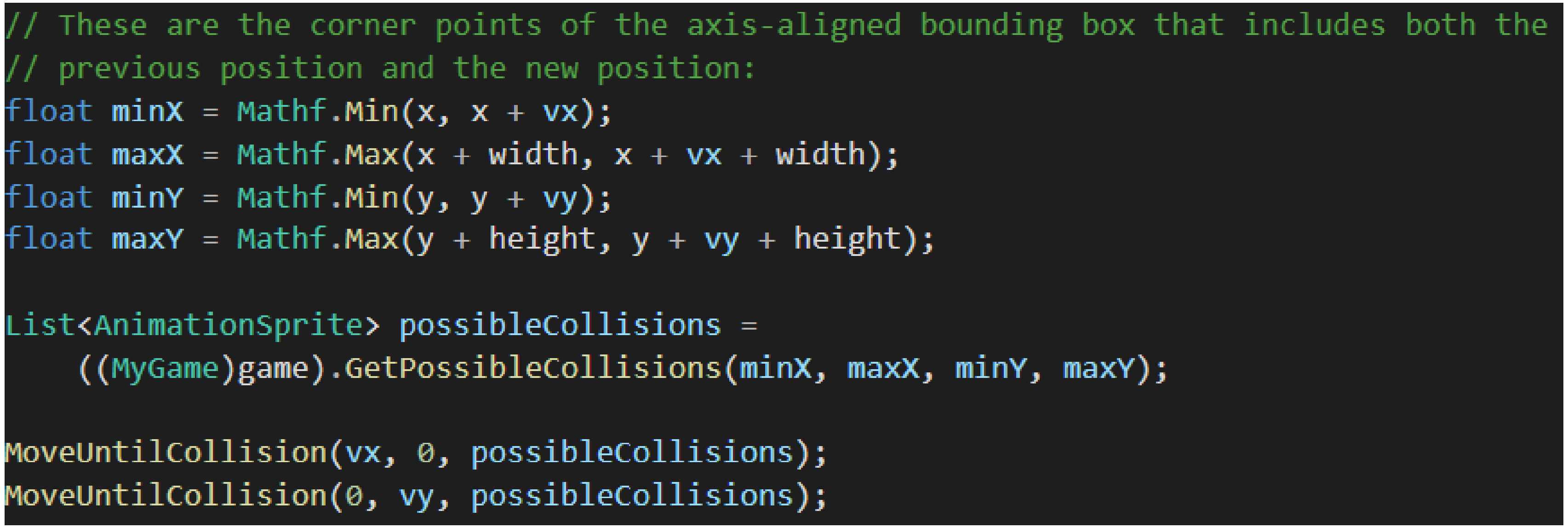
- By default, GetCollisions and MoveUntilCollision check against *every* collider
- Even if you have a 100 x 100 grid, if your player sprite overlaps only with 2 columns and 2 rows: you actually only need to check for 4 possible collisions!
- Main idea:
  - Store all tiles in a *2D array of Sprites* (*null* if no tile at that position)
  - Based on player position, you can get a short list of *collision candidates*
  - For all possible collision candidates:
    - Instead of GetCollisions, use HitTest
    - MoveUntilCollision: pass an optional list of colliders-to-check-against



Only four possible positions where a current collision tile could be

Space Partitioning

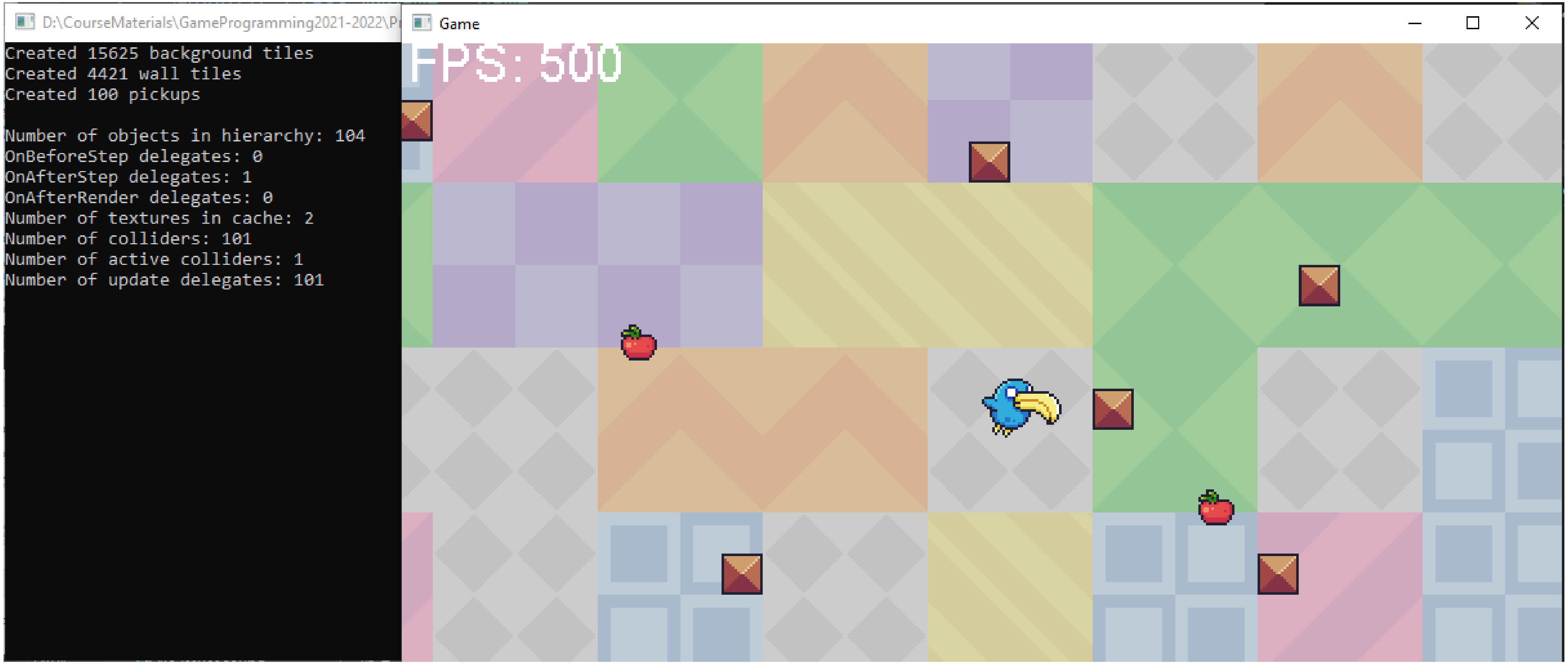
- See the Space Partitioning code sample on Blackboard
- *Demo*



```
// These are the corner points of the axis-aligned bounding box that includes both the
// previous position and the new position:
float minX = Mathf.Min(x, x + vx);
float maxX = Mathf.Max(x + width, x + vx + width);
float minY = Mathf.Min(y, y + vy);
float maxY = Mathf.Max(y + height, y + vy + height);

List<AnimationSprite> possibleCollisions =
    ((MyGame)game).GetPossibleCollisions(minX, maxX, minY, maxY);

MoveUntilCollision(vx, 0, possibleCollisions);
MoveUntilCollision(0, vy, possibleCollisions);
```

- Advanced: if sprites are not rotated, you can even only store the tile numbers in a 2D int array, and do your own collision checking logic (without using colliders!)

Fully Optimized



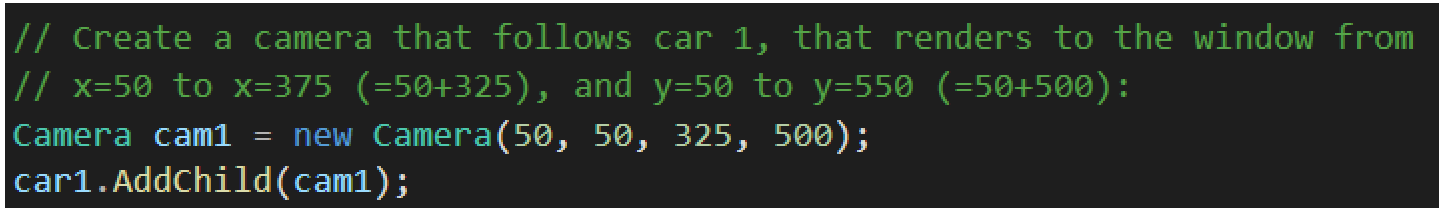
```
Created 15625 background tiles
Created 4421 wall tiles
Created 100 pickups

Number of objects in hierarchy: 104
OnBeforeStep delegates: 0
OnAfterStep delegates: 1
OnAfterRender delegates: 0
Number of textures in cache: 2
Number of colliders: 101
Number of active colliders: 1
Number of update delegates: 101
```

Space Partitioning Summary

- This grid-based space partitioning technique is a type of *broad phase collision checking:* first do a quick check to find all the possible collisions, then do a detailed check.
- There are other techniques as well, which also work if you don’t have a grid-like structure, or a *humongous* but *sparse* level
- Example: *quad trees*
- Outside of the scope of this course!

Cameras

Camera



```
// Create a camera that follows car 1, that renders to the window from
// x=50 to x=375 (=50+325), and y=50 to y=550 (=50+500):
Camera cam1 = new Camera(50, 50, 325, 500);
car1.AddChild(cam1);
```

- In the AddOns folder, you can find a *Camera* and *Window* class
- You can create a Camera (GameObject), which creates a rectangular window to render to.
- If the camera is in the hierarchy: it renders to its window *after* the main render loop
- Scale the camera to zoom, move the camera to scroll, etc.
- Use cases:
  - Mini map
  - Split screen
  - Alternative scrolling (including possibly rotation)
- *Demo* (see also the code sample on Blackboard!)

Camera: Tips and Pitfalls

- When using a *full screen* camera (for scrolling): set *game.renderMain*=false, otherwise everything is rendered twice! (But: don’t forget to set it to true again once the camera is destroyed!)
- Combining cameras with UI (HUD) can be tricky. Possibilities (hints):
  - *Easiest:* Draw HUD elements outside of camera windows (main render loop)
  - Make HUD elements child of camera (works well with *one* camera)
  - Subscribe to game.OnAfterRender to render the HUD manually
  - ***Add a “HUD Camera”, which is rendered last, with clearBackground=false*** (see the camera code sample on Blackboard)
- There are pros and cons to each approach → experiment!

Working with Files and Settings

Settings: Why

- It’s good to have *dynamic settings* that can be changed without changing or rebuilding the project
- Examples:
    - Players want to change the *keyboard input* (e.g. arrows vs WASD)
    - Different project team members have *different screen resolutions*
    - One team member works on art (full screen), while the other works on debugging (windowed, console visible)
    - The lead game designer in the project team wants to quickly *tweak game play values* without working in Visual Studio
    - You want to enable *modding*: loading different, user created levels
- The *Settings* class (in AddOns) enables this

Settings: How

- Add a file *settings.txt* to bin/Debug
- It should contain variable value assignments that exactly match variables in Settings.cs (name and type)
- Feel free to extend Settings.cs with more (public static) variables!
- Call *Settings.Load()* (typically before creating and starting your game)
- *Demo* (See also the example on Blackboard)

Full Screen and Exceptions

- When the game is ready for release, you probably want to run it full screen
- If this triggers an *exception*, this may leave the program frozen(!)
- It’s best to *catch exceptions* when running full screen / in release mode
- On the other hand, while debugging, it’s helpful to let your IDE show where the exception occurs

- Good setup: see the SettingsDemo on Blackboard *(demo)*

Working with Files

- You may want to store *persistent data*, that stays when the program closes:
  - Save games (progress, customization)
  - High score tables
  - User settings
- C# offers many tools for this, in the *System.IO* namespace
- *StreamReader / StreamWriter:* for working with text files
- *BinaryReader / BinaryWriter:* for working with binary files
- Both wrap a *FileStream*
- See the demos on Blackboard

Text Files and String Parsing

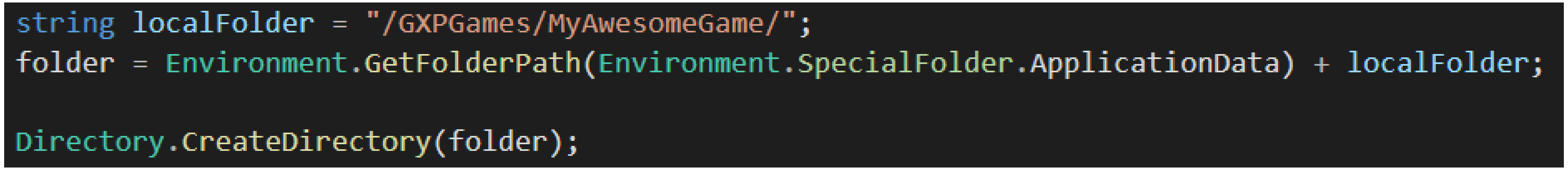
- Working with text files (StreamReader / StreamWriter): requires *string parsing methods*
- Often used methods: *Split, IndexOf, Substring, int.Parse, float.Parse*
- See the *demo* on Blackboard

Tips

- Think about your `file format’:
  - what do you really need to store?
  - How to make it robust? (e.g. when also doing manual editing)
  - How to make it future proof? (when adding data, can you still read old files?)
- When working with file I/O and string parsing: always include exception handling!
- Properly dispose of your resources (close the reader/writer, or use a “using” block)

Text Files vs Security

- It may seem weird to store save games and high scores in plain text
- Nevertheless, I still recommend it, because:
  - It’s actually really convenient to be able to view and edit it ☺
  - True security doesn’t exist anyway, when it’s all on the user’s machine
  - Who cares when someone `cheats’ on their own machine – they’re only spoiling their own experience (or maybe even improving it…?)
- If you want, you can come up with some *obfuscation* method or use a BinaryReader/Writer (but again: true security doesn’t exist)
- You can also store (hide) your file somewhere else:



```
string localFolder = "/GXPGames/MyAwesomeGame/";
folder = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + localFolder;

Directory.CreateDirectory(folder);
```

Assessment

A Look at the Grading Criteria

- Do code reviews (during or outside of the lab) – this will tell you where you currently stand
- Grading categories:

• Game play	(Every lecture, especially Lec. 2)
• Code quality	(Most lectures, especially Lec. 2)
• Software architecture	(Lec. 3: Encapsulation. Lec. 4: Inheritance, …)
• User feedback	(Lec. 4: HUD. Lec. 5: VFX/SFX)
• Tooling	(Lec. 3: Level loading. Lec. 6: Settings & files)

Summary of lectures

1. *Introduction:* Getting started, sprites, input, transformables
2. *Game Objects:* Hierarchy, collisions, type checking and casting, create & destroy
3. *Level Loading:* Managing complexity (encapsulation), arrays & lists, Tiled, linking objects, keeping information
4. *User interface, inheritance:* EasyDraw, events, correct level switching, virtual & override, Object Oriented design
5. *VFX & SFX:* Sounds, particles, tweening, blend modes, math (functions)
6. *Advanced topics:* Sharing your work, optimization, camera, files

Assessment

- Sign up for the assessment in advance (see upcoming Blackboard announcements!)
- An assessment schedule will be posted
- Upload your project before the assessment
- Having a working project with certain minimum features is *required*, but…
- *…You will be graded on your understanding / how well you can explain your choices and code* (it’s not just about having features!)

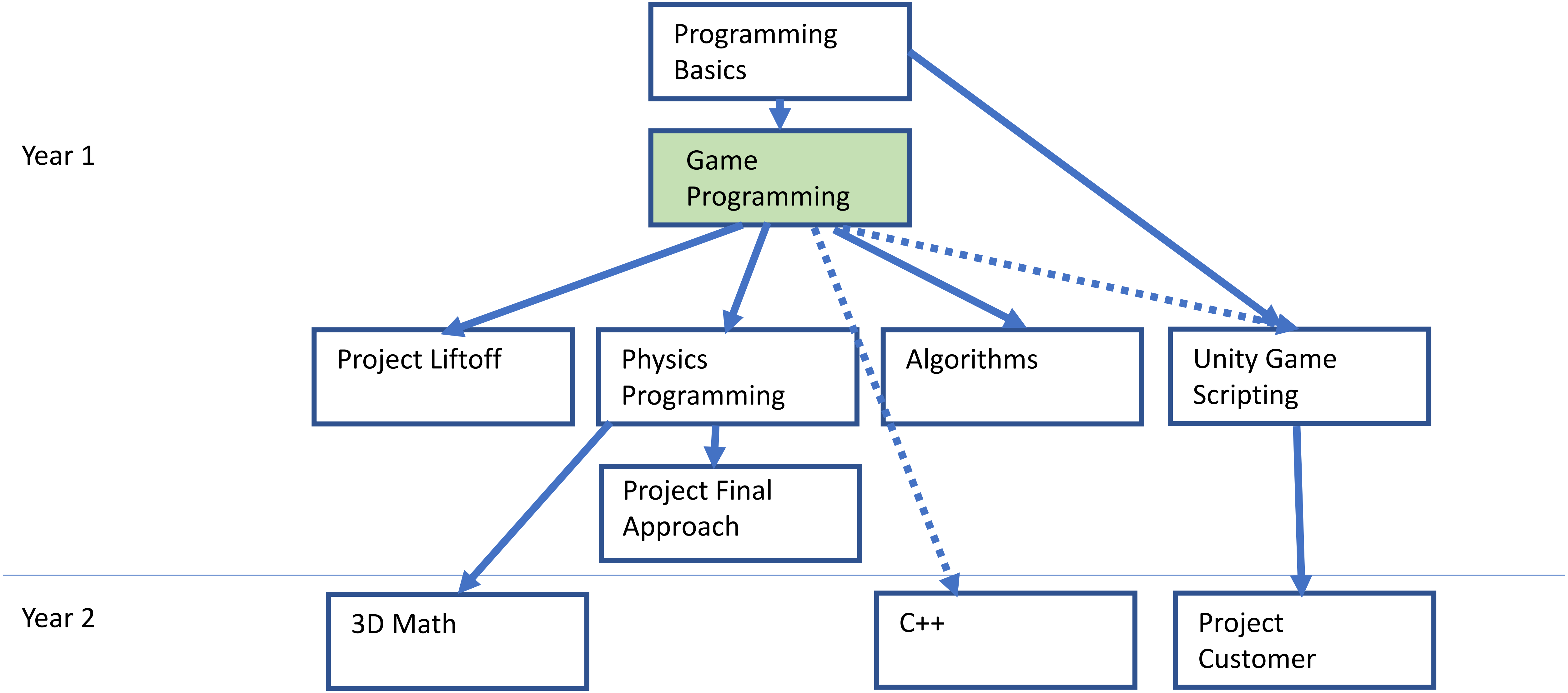
Outlook

- Actually, passing the upcoming assessment is only the first *(and easiest)* step in your journey to become a CMGT engineer / game programmer
- Make sure to *save* and *revisit* these lesson materials later! (Especially if you pass this course with just a 6 or 7…)
- This course was not about “learning the GXPEngine” – 90% of the knowledge is useful in almost any future (game) programming context

Connection to other Courses

→ Required knowledge

⇢ Useful knowledge



Project Liftoff