

CMPM 120

---

# Assets, Collisions, Transitions...

---

# Activate Transcription

# Questions

---

<https://forms.gle/FPte5yUatVADbxVS5>

# Schedule Overview\*

- 6/22      Introduction
- 6/24      Programming Our First Phaser Game
- 6/29      Version Control & Debugging
- 7/1        Scenes, Loops, Physics
- 7/6        Input, Cameras, State Machines
- 7/8        Assets, Collisions, Transitions**
- 7/13      JSON, Tilemaps, Map Editors
- 7/15      Tweens & Particles
- 7/20      Special Topics
- 7/22      Final Presentations

\*This will inevitably change a bit

# Schedule Overview\*

6/22	Introduction			
6/24	Programming Our First Phaser Game	Rocket Patrol Tutorial Due	6/26	
6/29	Version Control & Debugging	Rocket Patrol Mods Due	6/29	
7/1	Scenes, Loops, Physics			
7/6	Input, Cameras, State Machines	Endless Runner Due	7/6	
7/8	Assets, Collisions, Transitions			
7/13	JSON, Tilemaps, Map Editors	Final Game: First Build	7/13	
7/15	Tweens & Particles			
7/20	Special Topics			
7/22	Final Presentations	Final Game Due	7/22	

\*This will inevitably change a bit

# The Week Ahead

- Tuesday, Jul 6
  - ◆ Endless Runner [~20-30 hours] due by 9am
- Thu Jul 8, 2021
  - ◆ Final Game Theme Announcement: 9am
  - ◆ Console API due by 9am
- Fri Jul 9, 2021
  - ◆ Game Programming Learning Styles due by 11pm
- Tue Jul 13, 2021
  - ◆ Final Game: First Playable Build [~15-30 hours] due by 9am
  - ◆ Game Cameras due by 9am

◆ Remember that the deadlines for the Readings are suggested times when they would be helpful to know, not hard requirements!

# If your Endless Runner isn't finished

I expected that some of you wouldn't be done with it yet. Here's what to do with an unfinished project:

- Before the end of the grace period:
  - ◆ Please submit what you have with a note telling us that it is currently incomplete
- After the end of the grace period:
  - ◆ Email Isaac (at [ikarth@ucsc.edu](mailto:ikarth@ucsc.edu)) with the links to your repository and the web page where I can play the game.
  - ◆ Include a description of exactly what changed since the Canvas submission.

This applies to all projects! (Except the final presentation)

This class is intended to measure completeness, not speed!

# Questions

Changing which animation a sprite is playing

Smarter AI enemies

Custom Fonts

Enemies that fire projectiles

How to make a level for a platformer

How to animate tileSprites

# Assets

---

# Art Data Assets

- Some benefits:
  - Artists can shape the player experience without touching (much) code!
  - ...
  - Wait, what's the alternative? [Procedural art](#).
- Some drawbacks:
  - Loading time breaks immersion and flow.
  - Energy usage while loading discourages mobile play.
  - Large downloads may cause players to hit data transfer limits or storage limits, discouraging play even on desktop.
- Design considerations:
  - Can I delay loading assets that might not be seen? Most people only see the first few minutes of a game anyway.
  - Can I make my asset files smaller (in bytes) without hurting quality much?
  - Can I bundle assets together for efficient loading?
  - ...
  - How hard is it for artists to change a single pixel and see the result in-game?

# Aside: Mobile energy (battery usage) usage



**Radio communication** (downloading assets; multiplayer data)

**Processor usage** (showing distracting animations; running game logic)

**Screen brightness** (a white LED flashlight with a colored LCD in front)

Great mobile experience for podcast apps:

- Data is loaded in bulk (not many tiny segments) and then radio can go quiet.
- Data is saved between sessions (no re-downloading).
- Start listening right away (while download continues).
- Screen goes dark.
- CPU wakes up only rarely to decode a segment of audio before going right back to sleep.

You can listen to podcasts for days on a single charge!

Poor mobile experience for in-browser games:

- Data loaded in many chunks, each involving separate network transactions; radio active for a long time.
- Data rarely cached between sessions.
- Player watches animation on loading screen with almost full brightness and CPU usage.
- Screen never goes dark while playing.

Your battery is dead in an hour! (Installable apps improve the experience somewhat, but they don't address processor/screen costs.) *Could you make an audio-only game?*

# In the olden days...



Game cartridges for consoles used to contain (still do!) memory chips.

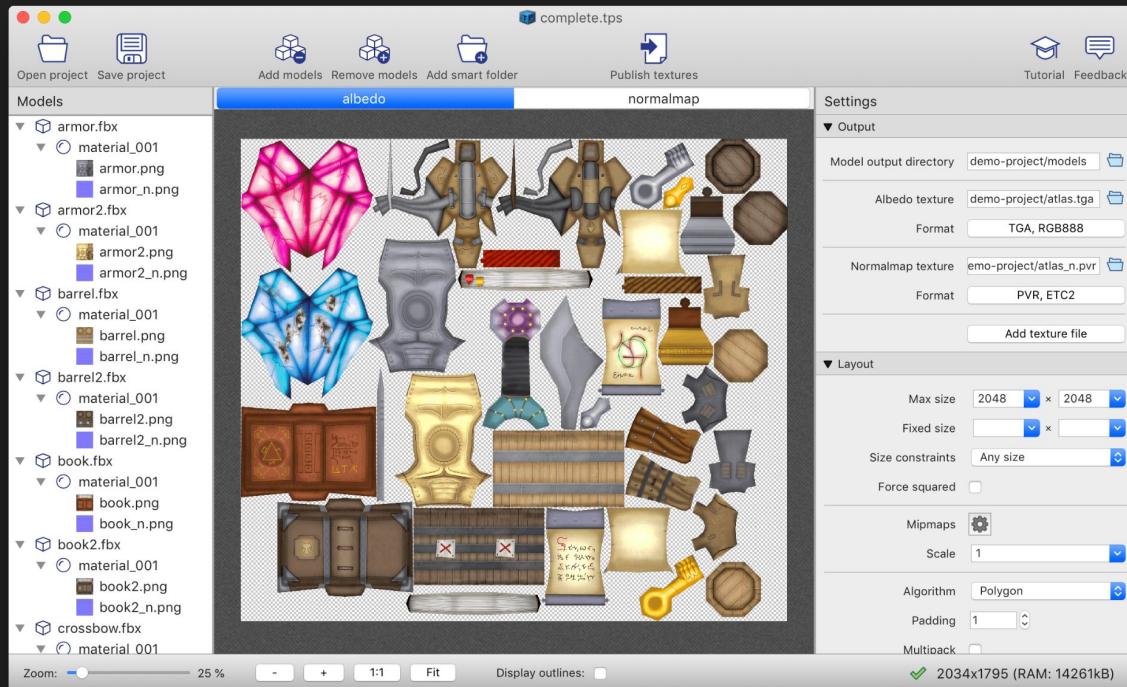
Pros:

- Very fast load time!

Cons:

- Expensive to manufacture/ship
- Player must wait for physical stock in store to buy
- Tedious for artists to update assets during development

# In the present day... (assuming you aren't making custom carts)



We have flexibility about loading many small individual assets or bundling them together for efficient loading using advanced tools. (Using “texture atlases” and other bundles.)

## Pros:

- More efficient use of radio+processor
- Better memory use (wider audience with older devices)

Major con: Artists must re-run the bundling tool to produce the files used by the game.

Metaphor: Is it faster to walk, bike, drive, train, plane? It depends on the destination and frequency of trip.

<https://www.codeandweb.com/texturepacker3d>

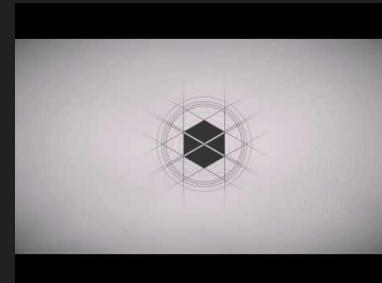
# Hiding your loading time



Sometimes you can hide the loading time by making your loading screens feel a little more like the gameplay screens, but most players will see right through this.

Trouble: When do you load the assets that are used to play the loading screen???

Sometimes simple procedural graphics are best.



# Assets in the *Phaser 3* framework

---

# “Notes of Phaser 3”

The screenshot shows a web browser window with the title bar "Notes of Phaser 3". The address bar contains the URL "rexrainbow.github.io/phaser3-rex-notes/docs/site/index.html". The page itself has a blue header bar with the title "Notes of Phaser 3" and a search bar. On the left, there's a sidebar with a navigation menu for "Notes of Phaser 3" containing various categories like Home, Phaser, Links, Phaser3, etc., each with a dropdown arrow. The main content area is divided into sections: "Home" (with a "Phaser" section), "Links" (with a "Phaser3" section), and "Table of contents" (listing Phaser, Phaser3, and Rex plugins). The "Phaser" section in the Home area contains a brief description of Phaser as a 2D game framework.

Notes of Phaser 3

Home

System ▾

Loader ▾

Game object ▾

Input ▾

Audio ▾

Camera ▾

Tween ▾

Shader effects ▾

Logic ▾

Data ▾

Math ▾

String ▾

Time ▾

UI ▾

Board ▾

Firebase ▾

Home

Phaser

Phaser is a fun, free and fast 2D game framework for making HTML5 games for desktop and mobile web browsers, supporting Canvas and WebGL rendering.

Links

Phaser3

- FAQ
- Official discord channel
- API document
- Examples
- Bug report

Table of contents

Phaser

Links

Phaser3

Rex plugins

An excellent and fast-loading reference for Phaser 3: (bookmark it)  
<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/index.html>

# A few of Phaser's loadable data types

- Images (good for static images)
- Sprites (moving, animated game objects)
- Sprite Sheets (a grid of tiled images)
- Texture Atlases (flexible collection of named sub-images)
- ...
- Audio (single audio file)
- Audio Sprites (a collection of name sub-clips)
- ...
- Video
- ...
- Binary Data (any other raw assets you want, TOML/YAML files?)
- Scenes

<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/loader/#file-types>



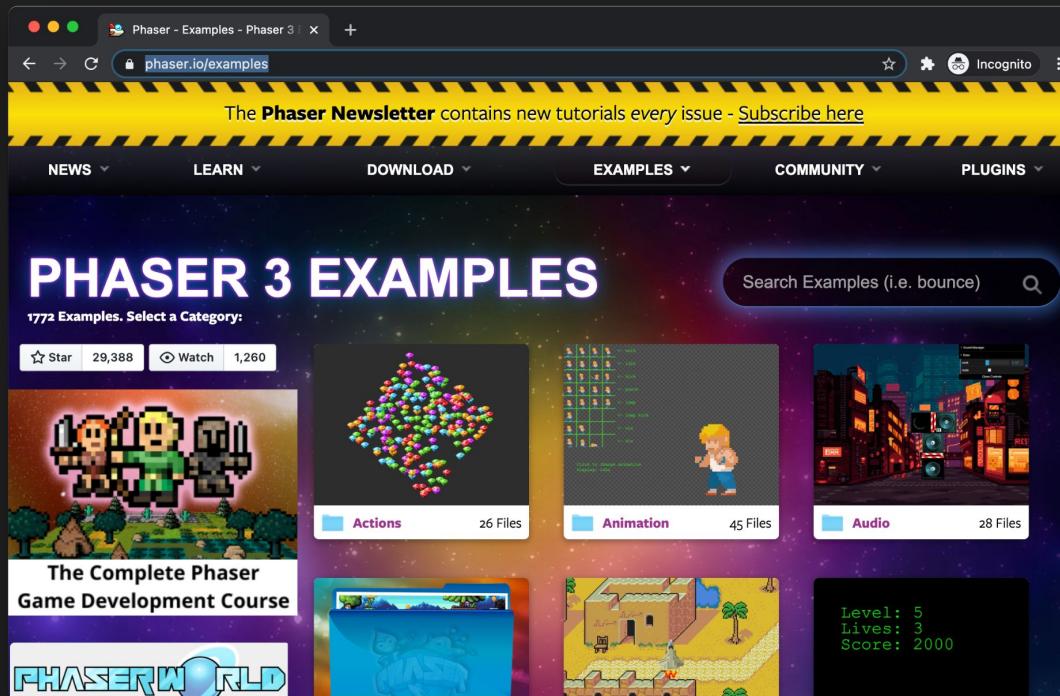
# General asset usage pattern

```
class MyScene extends Phaser.Scene {  
    preload() {  
        this.load.image("ship", "assets/ship.png");  
    }  
    create() {  
        this.myShip = this.add.sprite(10, 20, "ship");  
    }  
}
```

file path on your computer

key / nickname (don't need to match filename)

# Phaser's searchable example index



Browse: <https://phaser.io/examples>

Source: <https://github.com/photonstorm/phaser3-examples/tree/master/public>

# Nathan's example projects for CMPM 120 specifically

<https://github.com/nathanaltice?tab=repositories>

Let's clone and inspect **AssetBonanza**.

Plan:

- Let's look at where the assets are being loaded in the Network devtools page.
- Let's display many many more assets and look at effects on FPS meter.

# What can I do with a game object (e.g. Sprite, etc.)?

<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/gameobject/>

[everyone: open this page and research what you can do with a game object]

# Advanced questions

- Can I show stuff on the screen while assets are loading? (like in *Destiny 2*)
  - <https://phaser.io/examples/v3/view/loader/loader-events/load-progress>
- Can I use an asset on the loading screen? (like in *Mass Effect* elevators)
  - <https://phaser.io/examples/v3/view/loader/loader-events/play-animation-during-load>
- Can I start play before the assets are loaded and have them pop in when ready? (like *Mass Effect* core gameplay)
  - <https://phaser.io/examples/v3/view/loader/loader-events/display-file-as-loaded>

## If time permits...

- Let's check out the Leshy SpriteSheet tool for making texture atlases:  
<https://www.leshylabs.com/apps/sstool/>
- Let's talk about procedural graphics in Phaser:  
<https://phaser.io/examples/v3/view/camera/graphics-landscape>

# Break

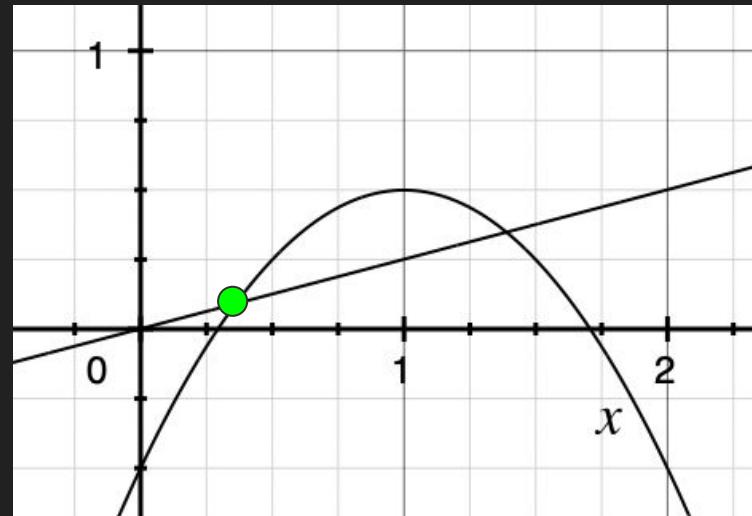
---

# Collisions in Detail

---

# What do we mean by “collisions” in game programming?

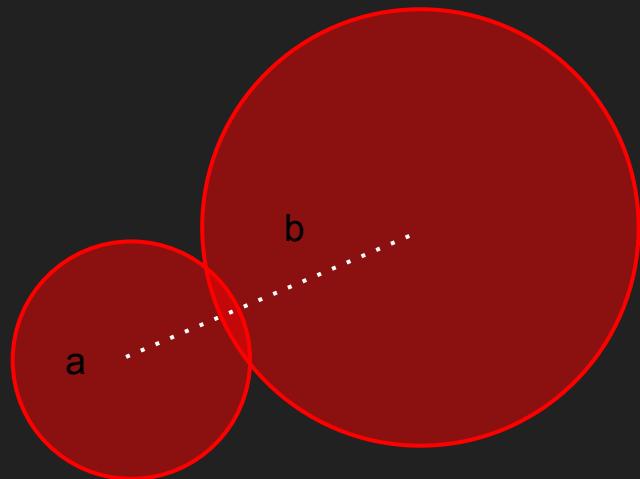
- **Collision detection:** Are two objects coincident in the world right now?
- **Determination:** Precisely when and where did they **first** come into contact?
- **Resolution:** If one object should just pass through another, how should the collision be resolved?



*Geometrically, do the lines intersect?  
Algebraically, does the system of equations have any solutions?*

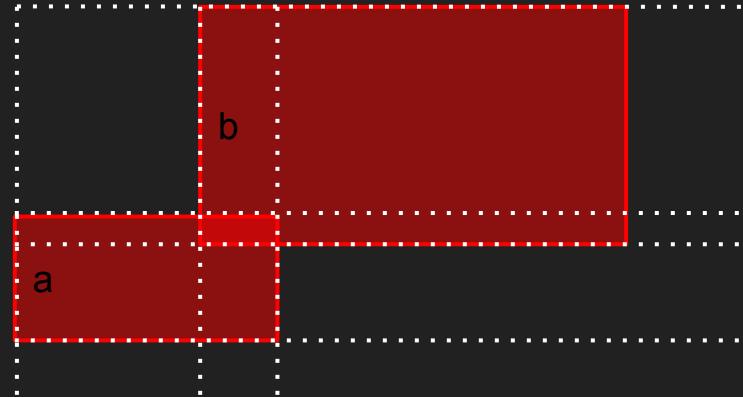
*Which solution has the **least** x value?*

# Collision detection between two simple shapes



**Circle versus circle:** Is the distance between the center of the circles less than the sum of the two circles' radii?

```
dist(a.center,b.center) < a.radius + b.radius
```

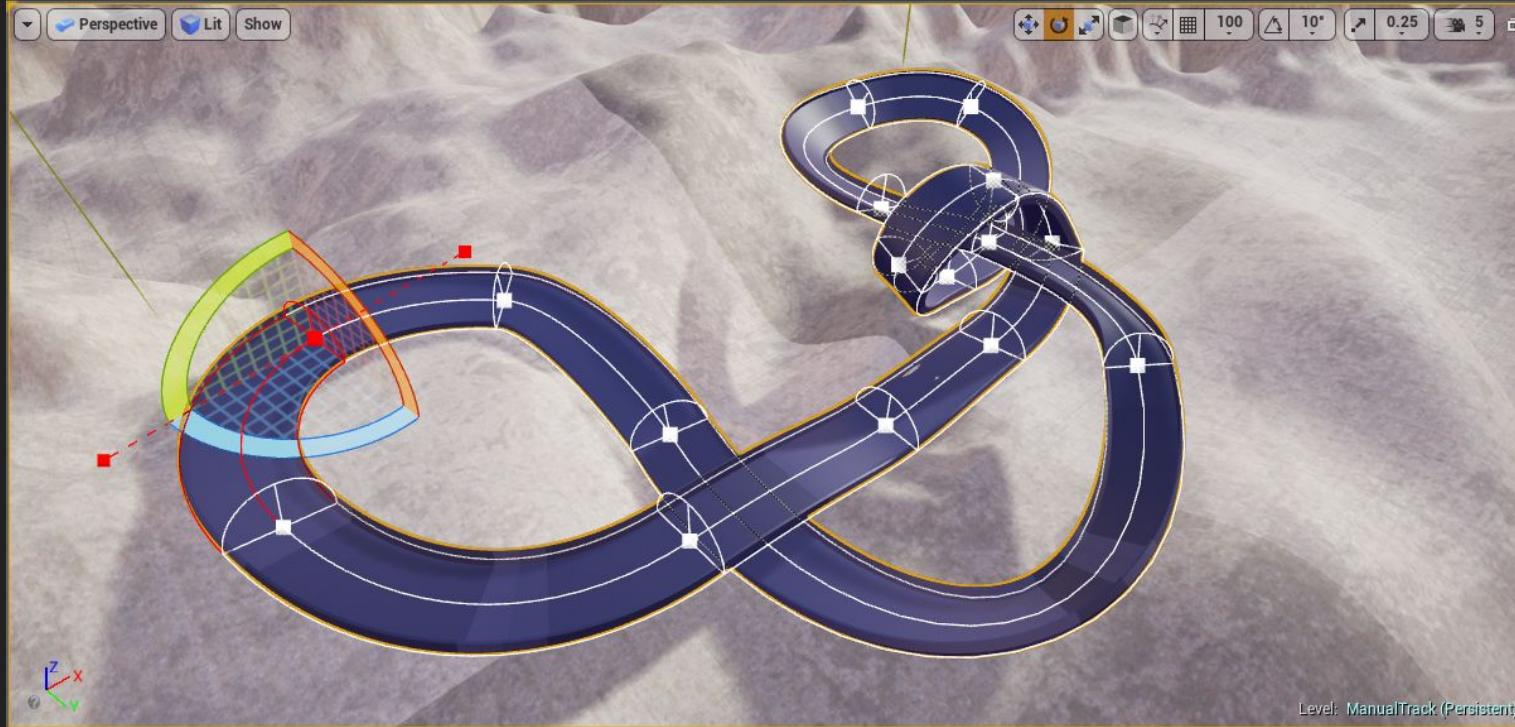


**Axis-aligned rectangle versus rectangle:**

```
min(a.x,b.x) + min(a.width,b.width) >  
max(a.x,b.x) && min(a.y,b.y) +  
min(a.height,b.height) > max(a.y,b.y)
```

(or something like this)

# 3D point versus expanded 3D spline?



The algebra is trickier (finding roots of higher-order polynomials), but it boils down to just one ugly line of code.

Think of the **quadratic formula** you memorized in middle school.

# Pixel-precise sprite versus sprite collision detection



We could find the range of pixels where the bounding boxes for each sprite overlaps (using min/max logic from previous slide). Then we could **loop over every pixel location**, checking if both sprites have a non-transparent pixel in the same location at the same time.

This can get inefficient...

# Pixel-precise collision in *Montezuma's Revenge* (1984)



The Atari 2600 / Video Computer system had a ~1MHz, 8-bit processor.

How could games for this platform implement pixel-precise collision detection in 1984???

# Let's crack it open at 8bitworkshop

Play with pixel-precise collision mechanics:

<https://8bitworkshop.com/v3.7.1/?platform=vc&file=bb%2Fsample.bas>

Edit line 47 (if the Batari Basic program) to slow down updates.

```
player0:  
%01000010  
%11111111  
%10000000  
%00000000  
%00000000  
%00000000  
%00000000  
%00000000  
%00000000  
  
COLUP0 = 75
```

```
player1:  
%00001000  
%00001110  
%00001010  
%00111110  
%00101000  
%11111000  
%10000000  
%11100000
```

```
COLUP1 = 153
```

```
update() {  
    if(displayHardware.didPlayfieldAndBallSharePixelsThisFrame) {  
        handlePlayfieldCollision();  
    }  
}
```

Single-step the assembly code for collision resolution:

<https://8bitworkshop.com/v3.7.1/?platform=vc&file=examples%2Fcollisions.a>

Start reading the (6502 assembly) code at line 212. Tweak the constant on line 215 to change ball return speed.

# Okay, so how did the hardware do it?

8bitworkshop's got you covered. Let's drop down into the hardware design:

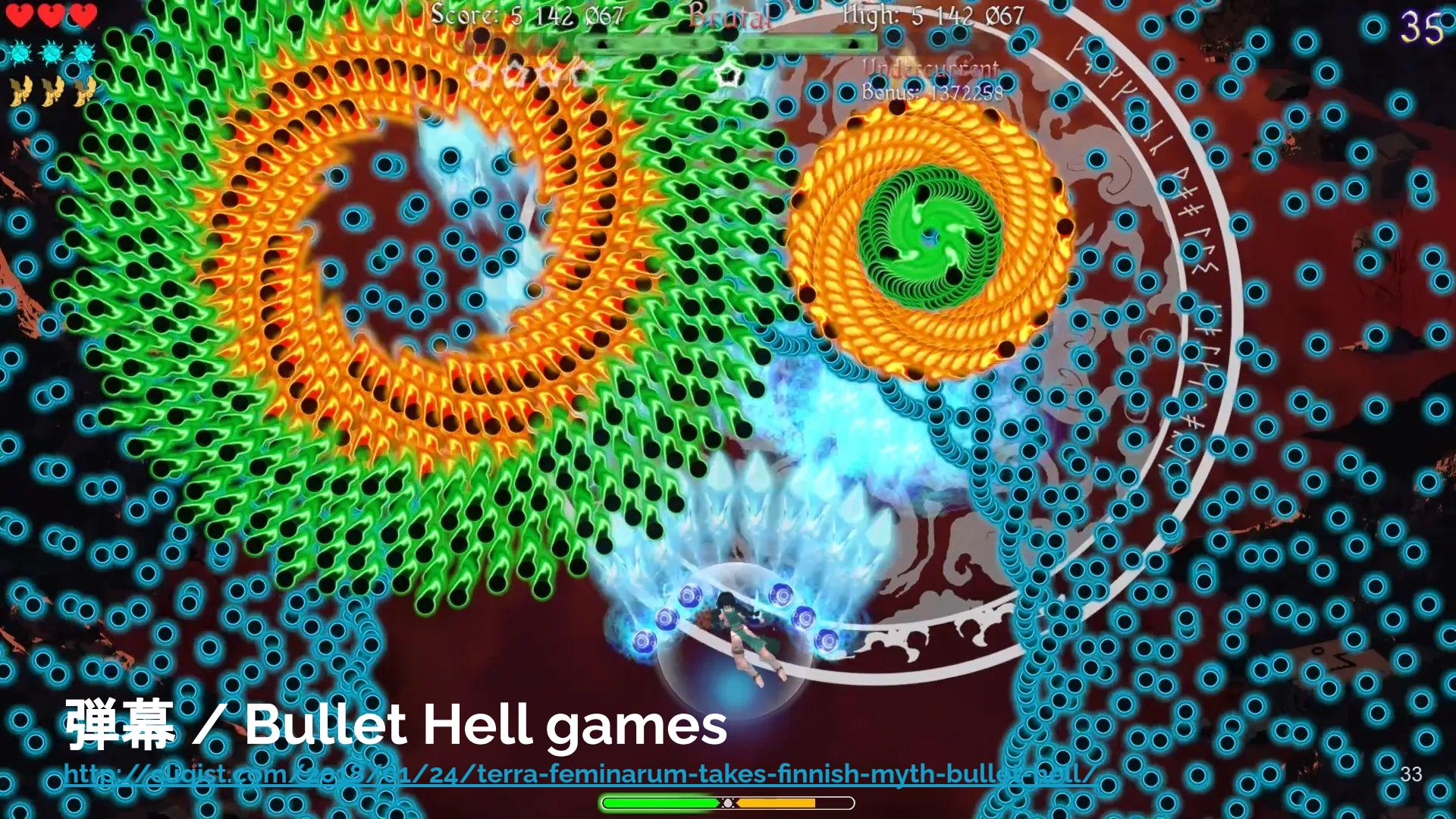
[https://8bitworkshop.com/v3.7.1/?platform=verilog&file=sprite\\_rotation.v](https://8bitworkshop.com/v3.7.1/?platform=verilog&file=sprite_rotation.v)

Line 247 checks collision continuously with an AND gate during signal generation.

Line 306 latches and resets this value.

The full set of callable objects is baked into the hardware design, and it can't be changed without a soldering iron and melting metal.

The Atari 2600 supported two **lines** (Player0 and Player1), **two** missiles, and one **ball**. If you wanted to simulate more objects, the programmer needed to move keep moving these objects around (sometimes many times per frame) to act as collision probes.



# 弹幕 / Bullet Hell games

<https://s0ci0t.com/e/e/21/24/terra-feminarum-takes-finnish-myth-bullet-hell/>

# Collision between massive groups

You can ask the physics engine to do it for you:

```
this.physics.collide(this.goodGuys, this.badGuys, this.handleGoodBadCollide)
```

But how is this implemented?

```
function collide(groupA, groupB, callback) {  
    for(let a of groupA) {  
        for(let b of groupB) {  
            if(checkCollision(a,b)) {  
                callback(a,b);  
            }  
        }  
    }  
}
```

If there are  $n$  objects in each group, this approach has running time  $O(n^2)$ .

What does it mean? Doubling the number of objects might make your game **four times** slower. You'll hit interactivity limits quickly as  $n$  grows.

# Strategies to improve collision detection efficiency

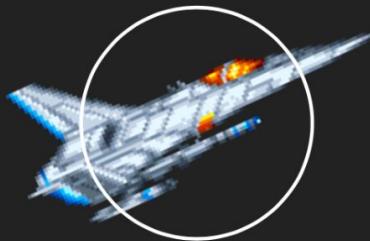
Reduce the number of pairs to consider:

- Only check for collision between groups that you care about
- Keep the groups small (or even limit them to one specific object)
- Summarize large groups by a single enclosing.

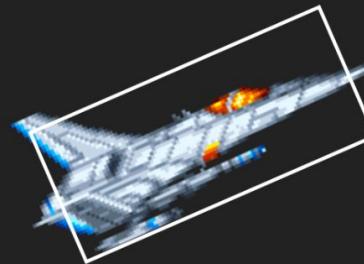
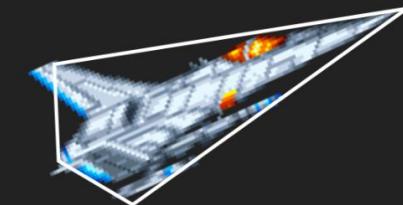
Reduce the complexity of individual collisions:

- Use simpler shapes to approximate more complex ones:
  - Pixels > triangles > convex hull > oriented boxed > aligned box > circle

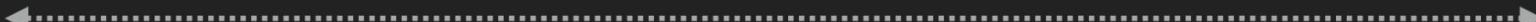
# Bounding volumes (summarizing complex objects/groups)



Circle/Sphere

Axis-Aligned  
Bounding BoxOriented  
Bounding Box

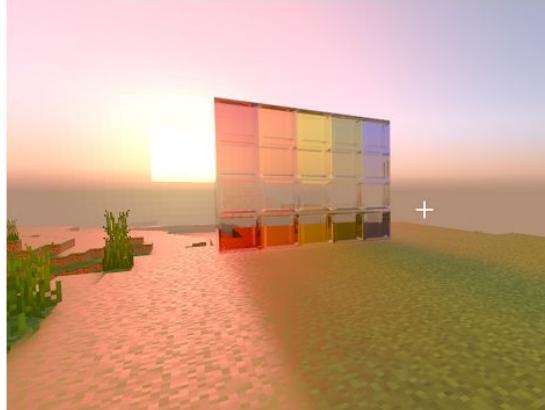
Convex Hull



Speed

Precision

# Aside: Real-time raytracing



Raytracing is an image rendering technique that involves many collision detection operations **per pixel** (to see which object a ray of light will hit next).

The outputs can be very impressive, but using them for **interactive settings** requires getting very clever about optimizing collision checks.

40 years later, we're still implementing parts of our rendering and game logic in hardware (in GPUs this time).



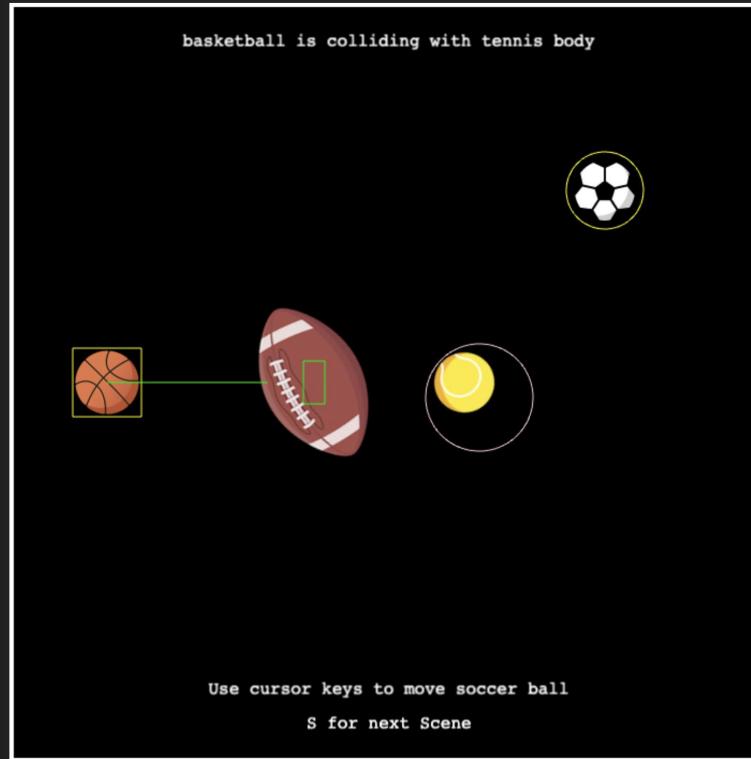
In the **broad phase** of collision detection, identify small and localized groups of objects that might collide using inexpensive/imprecise methods (such as spatial hashing).

In the **narrow phase** of collision detection, only use the expensive/precise methods (e.g. based on the separating axis theorem) to check small groups.

Implication artists/designers: *If your game slows down when you add more objects, ask an engineer to see if collision detection is the performance bottleneck (engineers: use a profiler). There may be some engineering tricks available that improve performance at scale with minimal impact on the game feel.*

# Let's check out Nathan's BigBodies example project

Use 's' to switch between scenes.



# Exit Slip

---

<https://forms.gle/XdceZe1RrNkZ>  
R7JR6

# Bonus Slides

---

# The Week Ahead

- Tuesday, Jul 6
  - ◆ Endless Runner [~20-30 hours] due by 9am
- Thu Jul 8, 2021
  - ◆ Final Game Theme Announcement: 9am
  - ◆ Console API due by 9am
- Fri Jul 9, 2021
  - ◆ Game Programming Learning Styles due by 11pm
- Tue Jul 13, 2021
  - ◆ Final Game: First Playable Build [~15-30 hours] due by 9am
  - ◆ Game Cameras due by 9am

◆ Remember that the deadlines for the Readings are suggested times when they would be helpful to know, not hard requirements!

# Final Project

---

**Themes will be announced on  
Thursday, 9am during AGPM 120**

---

# Final Game

<https://canvas.ucsc.edu/courses/44176/assignments/256903>

- a link to your game's GitHub repository
- a playable link to your game online (e.g. GitHub Pages, itch.io, etc.).

# Final Game: Rubric

Your final game is graded based upon the following technical and aesthetic criteria, using cake-based metaphors 🍰:

## ==THE BATTER==

- +5 The game runs/executes without critical errors or crashes. (Graders will use Chrome, so be sure your game works in that browser.)
- +5 The game includes a title screen, a credits screen, an "ending," and the ability to restart from within the game. (These criteria are judged relative to your specific game, genre, artistic tone, etc.)
- +5 The player can learn the controls from within the game, whether through a tutorial, instruction screen, or other diegetic means.
- +5 The game is playable to completion by a player of moderate skill. If your game is purposefully difficult and you're concerned that the grader won't be able to evaluate it properly, please provide a "grader mode" or debug menu that will allow us to see everything you've made.
- +10 Your project and code are well-structured and organized, including legible comments, appropriate data structures, sensible prefabs, meaningful variable names, logical scene structures, etc.
- +10 Your project has a well-maintained and updated GitHub page that shows meaningful contributions, commits, and milestones throughout the course of the project's history.

## ==THE BAKE==

- +25 Your game has artistic cohesion, i.e. the art, sound, typography, etc. reflect your stated aesthetic/experience goals, and your assets make sense together.
- +25 Your game has mechanical cohesion, i.e. the mechanics reflect your stated aesthetic/experience goals, the game feels good to play, and the mechanics are well-implemented.

## ==THE ICING==

- +10 Your game has that extra bit of polish, creativity, technical prowess, and/or originality that helps it stand out from other games. We use this as a grade "tilt" to reward games that we really enjoyed, that are bold and inventive, that align well to the class themes, that demonstrate strong technical skills, and/or went beyond the stated objectives of the assignment.

# Final Project: First Playable Build

<https://canvas.ucsc.edu/courses/44176/assignments/256902>

Every team's game will be different, but there are a few key elements we want to see present in your first playable build. It's OK if you have lots of bugs, temporary assets, wonky physics, etc. A playable build just means that we can do something in your game. We will check for:

- **Basic Scene structure:** You have defined Scenes (yes, plural) and some means to switch between them (e.g., via in-game actions, temporary key presses, etc.). (10 points)
- **Player interaction:** The player can interact with the game. You should have at least one primary mechanic operational. If you make a platformer, perhaps movement and jumping are implemented. If you make a narrative game, perhaps the dialog boxes are implemented. If you're making a hidden object game, perhaps the basic point/click verbs are implemented. Primary mechanics will vary from game to game. (10 points)
- **Temporary visual assets:** What good are core loops and interactions if there is nothing to look at? These should be assets \*your team\* creates, even if they are geometric primitives, quick doodles, or magazine cutouts. (10 points)
- **Temporary sound assets:** Don't leave decisions about sound until the end of the design process. Sound is integral to making a game feel "real." Make sure your game makes some noise. (10 points)
- **Code organization and hygiene:** Be sure you're using good software engineering practices, including version control (git/GitHub), code commenting, code encapsulation (Scenes, prefabs, etc.), and other quality-of-life features (e.g., indenting, logical variable names, etc.). It doesn't have to be perfect, but a grader should be able to look at your GitHub repository and understand how your code is structured and how it works. (10 points)

# Answers

---

# I want to read the slides!

These slides are available online, linked from the Syllabus:

<https://canvas.ucsc.edu/courses/44176>

## Class Resources

- Discussions: [Canvas Discussions](#)
- Class Discord: [Shared with AGPM 120](#)
- Lecture Slides: [on Google Drive](#) and [on GitHub](#)
- Lecture Videos: [Playlist on YuJa](#)
- Professor Nathan Altice's Code Examples: [on Github](#)
- Phaser 3 Examples Index: [Spreadsheet of concepts and projects](#)
- 120 Troubleshooting FAQ: [This page on Canvas](#)

## Scoring / Grading

**I want a place where I can ask questions and see questions from other students!**

Canvas Discussions:

[https://canvas.ucsc.edu/courses/44176/discussion\\_topics](https://canvas.ucsc.edu/courses/44176/discussion_topics)

Class Discord (Shared with AGPM 120):

<https://discord.com/invite/AWPBTZt76r>

# Does it matter if something is lowercase or uppercase?

**JavaScript** is case sensitive

**Web Servers** depend on the operating system on the server:

- **Windows** is not case sensitive
- **Linux** and **OSX** are case sensitive

Therefore, it is possible for your website to behave differently on your local machine (if you are running Windows) and GitHub pages (which runs the web server on Linux)

**There wasn't enough time! I'd have liked a chance to do office hours between some of the parts.**

I would have preferred that too.

Since the class is so accelerated, please take advantage of the office hours even after the official due date of the assignment.

# Where is code going in Git?

Git keeps a local storage of all past commits in a hidden folder called `.git`

Mostly, you can ignore the hidden folder.

Once you've made a commit, you can always get that information back - you basically never have to worry about overwriting files!

Even once you've made a later commit, you can look at past files or restore them.

# **What are the guidelines for creating assets? What pixel size should the images be?**

Depends on the game - but in general you should aim for it to be the pixel size it will appear on screen.

Remember that the player has to download the images - so keep them small and don't make them bigger than they need to be!

# Does Phaser support a key being held down? What kind of input can I use in my game?

I'm glad you asked!

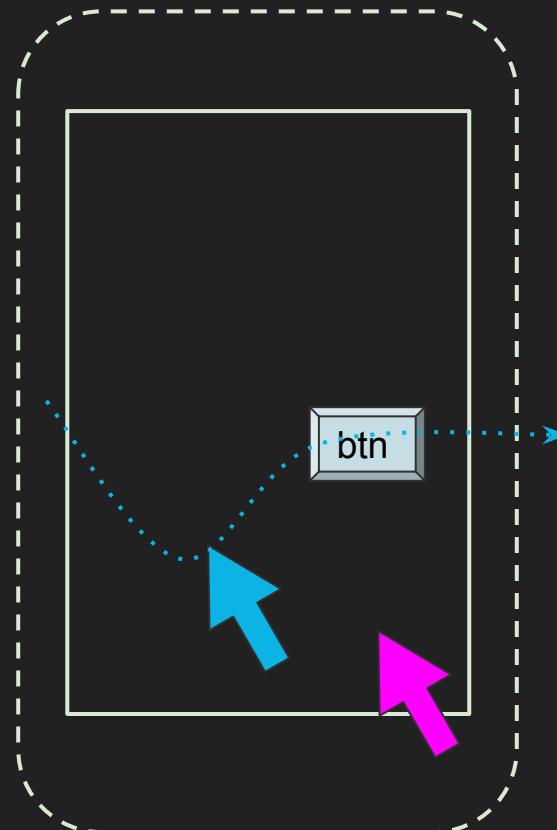
# Input and Movement

---

# Phaser-supported input types

- Pointers  (generalized fingers)
- Keyboard 
- Gamepad 
- Anything else in the browser that you want to rig up yourself:
  - Gyro/accelerometer 
  - Webcam 
  - Microphone 
  - Clock 
  - Network connection 
  - Page visibility 
  - Battery 
  - ... 

# Pointers



## Input events:

- down/up (click)
- move
- over (some object)
- out (of game canvas)
- Wheel (scroll)
- ...

# Nathan's *They Are Listening*

Let's check out (git clone) Nathan's example input demo built on the Phaser 3 game framework: <https://github.com/nathanaltice/TheyAreListening>

- **Scene-level** inputs
- **Object-level** inputs
- no `update()` for the scene in either case

Reference:

- <https://rexrainbow.github.io/phaser3-rex-notes/docs/site/touchevents/>
- <https://newdocs.phaser.io/docs/3.54.0/Phaser.Input.InputPlugin>

# Keyboard controls (seen in *Rocket Patrol*)

Methods:

- `this.input.keyboard.addKey( _ )`
- `this.input.keyboard.addKeys( {q: _, e: _} )`
- `this.input.keyboard.createCursorKeys()`
- ...
- `this.input.keyboard.on('keydown-A', callbackFunction, ...);`

Reference:

- <https://rexrainbow.github.io/phaser3-rex-notes/docs/site/keyboardevents/>
- <https://newdocs.phaser.io/docs/3.54.0/Phaser.Input.Keyboard.KeyboardPlugin>

# Custom input types

- Geolocation:  
<https://developer.mozilla.org/en-US/docs/Web/API/Geolocation/getCurrentPosition>
- USB:  
<https://web.dev/usb/>
- ...

```
let weirdSensorDevice = ...;

weirdSensorDevice.on(EVENT_NAME, function(event) {
    // TODO: do something with event
    // set some variables checked in update()?
    // call some other methods?
});
```

If you were making an endless runner that required the player to actually run outdoors with their phone, how would you teammates test the game during development? What if the developer or player device doesn't have GPS support?

Solutions: Cheat codes, alternative controls, accessibility modes, instructions up front, ...

(Make something that work using pointers/keyboards first, then use weird inputs to add depth to your game.)

# Two styles of input handling

Polling-based: ("check every frame")

```
update() {  
    console.log(keyA.isDown);  
    console.log(pointer.isDown);  
    console.log(pointer.x, pointer.y);  
}
```

Event-driven: ("call me when it's time")

```
create() {  
    this.input.keyboard.on('keydown-A', () => {  
        console.log('A went down!');  
    });  
    this.input.on('pointerdown', (pointer) => {  
        console.log('pointer went down!');  
    });  
}
```

Almost any game framework will support both input styles. You could use one to implement the other if it was missing. Which is nicest to use depends on the kind of interaction you are making.

# Input handling example projects

## Nathan's TheyAreListening project

Event-driven pointer input

Scene-level:

- `this.input.on(EVENT_NAME, fn)`

Object-level:

- `obj.setInteractive(OPTIONS)`
- `obj.on(EVENT_NAME, fn)`

## Nathan's LowKey project

Polling-based keyboard input

Once:

- `cursors =  
this.input.keyboard.createCursorKeys();`

Every frame:

- `if(cursors.left.isDown) {...}`

## Event-driven keyboard input

Once:

- `swap = this.input.keyboard.addKey('S');`
- `swap.on('keydown', () => {  
...  
});`

# What's *Phaser.Input.Keyboard.JustDown*?

Do you want to do something every frame while button is down? Just check `key.isDown`.

Do you want to handle the moment that the key was first pressed down?

Option 1: add some extra variable like `keyAlreadyPressed` to notice subsequent frames.

Option 2: use `Phaser.Input.Keyboard.JustDown(key)` in `update()`

Option 3: use `key.on('keydown', fn)` in `create()`

[Phaser internally uses the the first design to implement the second one.](#)

# Cheat codes speed up debugging

**Does testing a certain feature require playing the game for more than two seconds?** Does it require waiting for a timer to expire or require the player to reach a certain location? Does it depend on a lucky outcome of the random number generator? Does it require any player skill at all?

- Create your own cheat codes:
- Disable time limits
- Disable enemy damage effects
- Make character invulnerable
- Grant all special abilities
- Jump to specific story chapter
- ...

# Break

---

# Data

---

# Break

---

# State Machines

---

# What can the player do right now?

```
if(state == SITUATION_1) {  
    if(input == ACTION_1) {  
        // do thing A  
    } else if (input == ACTION_2) {  
        // do thing B  
    } else if (input == ACTION_3) {  
        // do thing C  
    }  
} else if (state == SITUATION_2) {  
    if(input == ACTION_1) {  
        // do thing D  
    } else if (input == ACTION_2) {  
        // do thing E  
    } else if (input == ACTION_3) {  
        // do thing F  
    }  
} else if ...
```

Adding more distinct situations and actions (and modeling the complex relationships between them) can lead to code that grows and grows.

If “[a game is a series of interesting choices](#),” are we doomed to write **messy** code when writing **interesting** games???

# The Quiet Room (2020)

## The Quiet Room



This conspicuous orange switch ought to be useful, but the cabling is too complicated for me to trace the source or destination of the power it controls.

- I cautiously flip the switch.
- I look for something else of use in the garage.

Here's a photo-based linking game Adam made as a code example for CMPM 35. In it, the player can do a few distinct things in many different situations. The transition logic for each situation is unique, so there's no immediately obvious way to reuse code between situations.

If-else hell ensues, right?

# Designer sketch for *The Quiet Room*



Carpet

Handle

Exterior



Knobs are useless,  
but also so good!

Writhing

Give up, give in.

Aftermath

Wtf????

Toggle the power on/off.



Garage  
Interior



Garage  
Door



Garden

Paint doesn't hum.



Alley



Yard

Bah, just chicken coop.



Coop

The story is inspired by a real audio art installation living in Adam's friend's backyard:  
<https://vessel.fm/2020/03/02/jet-sound-bath/>

## Let's capture state-transition logic with data rather than code!

Check out this cute duck game on Glitch (another CMPM 35 code example):  
<https://glitch.com/edit/#!/project-westlake>

Note:

- story content authored **story.yaml**
- see **startStory** and **startScene** in **engine.js**

# State Machines as a game programming pattern

A **design pattern** is a **reusable solution** to a commonly occurring **design problem**.

There are many game programming patterns, and **state machines** are one.

Using patterns to organize your game code can make it more readable and more maintainable. Knowledge of patterns can make you better at reading and maintaining code.

Among infinite ways to get a program to behave a certain way, we might prefer the ways that use familiar patterns.

# Cooking analogy

egg custard



yeasted dough



Knowledge of reusable patterns allows you to transfer knowledge from one design to another and helps others know where to inject their own variations.



custard pie



奶黃包



あんパン

# Movement code can get complex quickly!

Let's update() in Nathan's *MovementStudies*.

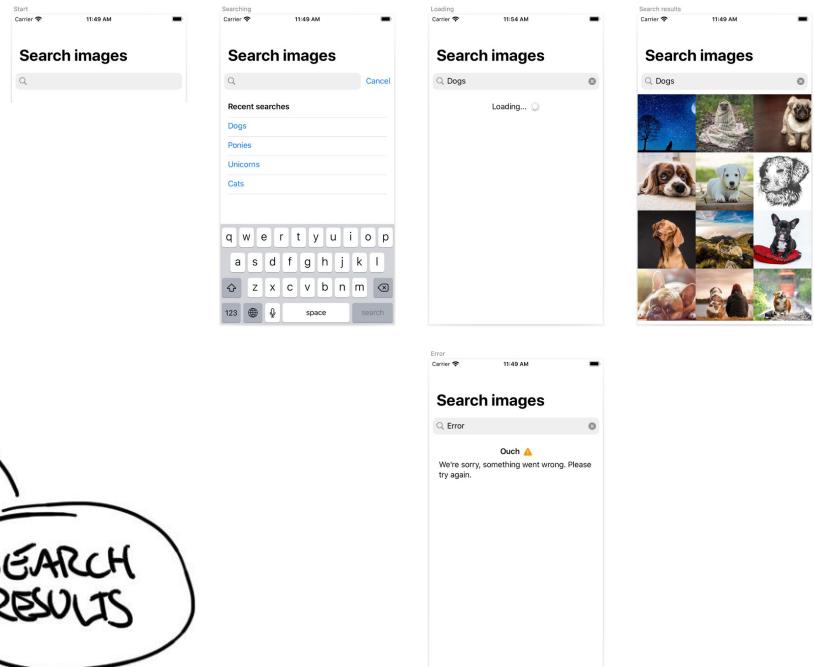
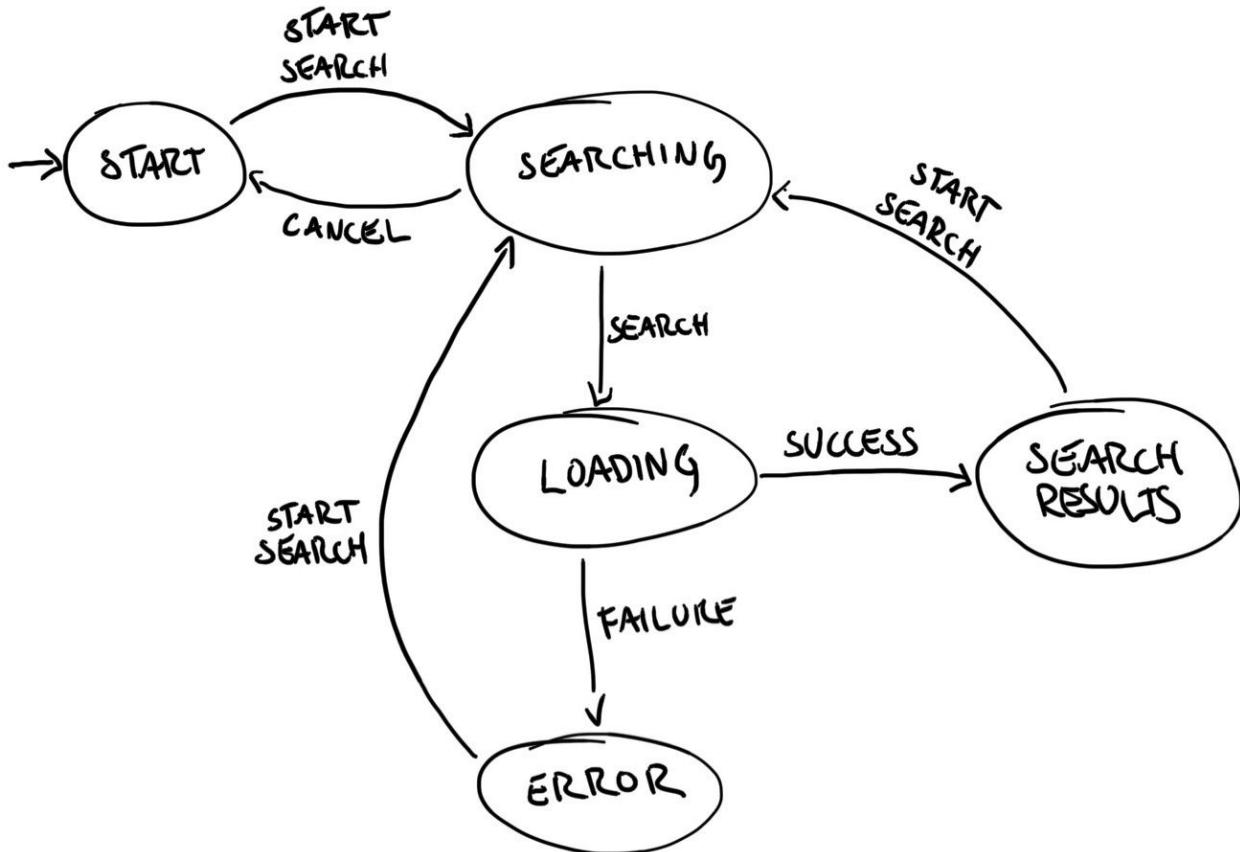
# Movement in *Celeste*

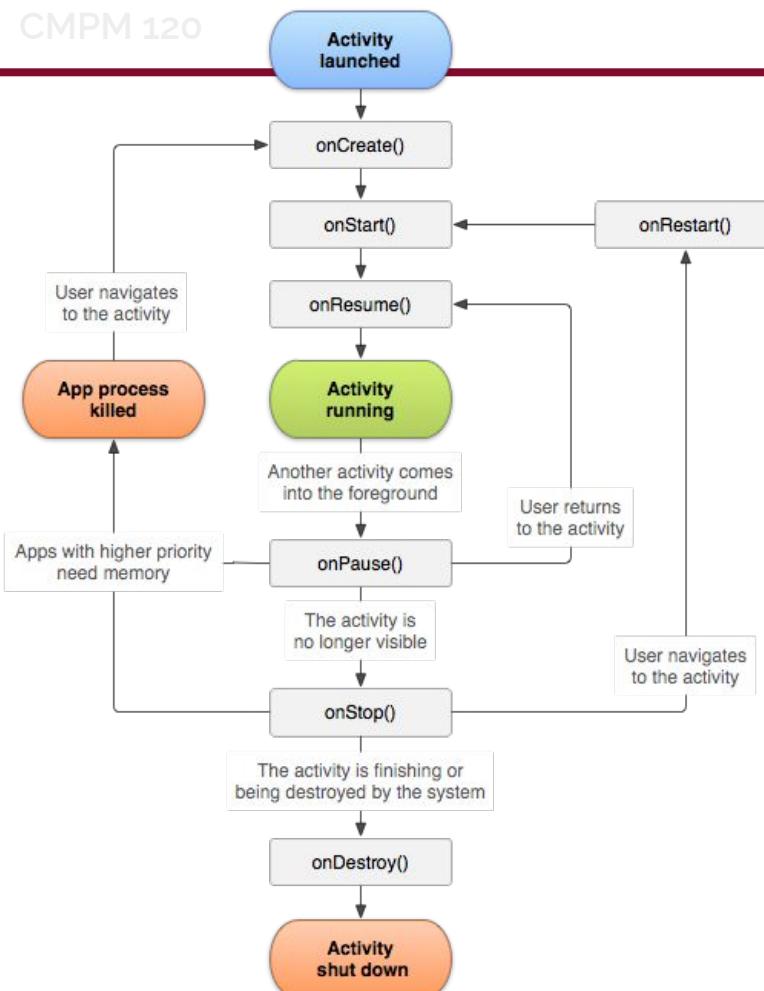


The Player class in Celete's C# implementation is mostly about handling movement, and it is over 5000 lines long!

<https://github.com/NoelFB/Celeste/blob/master/Source/Player/Player.cs>

It uses the **state machine** pattern to organize this necessarily-complex code for well-polished player movement.





In Android apps, Activity classes (like Phaser's scenes) encapsulate interactive situations.

Activity classes can have unique code in **methods** to handle starting, stopping, pausing, resuming, etc. Activities can launch/stack one another, but their code is separate.

# Let's check out Nathan+Adam's *SecretoftheOoze*

Notes:

- States and available events are modeled by a data structure (in JSON format).
- Because all states handle input identically (given data in state machine), no per-state code is needed.
- Public collaboration history:  
<https://github.com/nathanaltice/SecretoftheOoze/commits/master> (yay tweens)

# Let's check out Nathan's FSM example

Notes:

- States have unique logic captured in small State subclasses (in Hero.js)
- Clean update code:
  - ```
update() {  
    // handle hero actions  
    this.heroFSM.step();  
}
```
- Minor if/else messes contained within individual scenes (lets you diagnose and fix bugs locally).
- StateMachine class mostly hold current state and lets you transition to

# If time permits, see `dat.gui` in Nathan's Utilities

Notes:

- Tune magic numbers in your game without editing code or even restarting gameplay!

# Break

---

# Cameras

---

# Screen space versus World space

- [screen as world]
  - [Pong](#) (1969)
  - [Pac-Man](#) (1980)
- [camera onto mostly-2D world]
  - [Defender](#) (1981)
  - [Sonic](#) (1981)
  - [SimCity 2000](#) (1994)
  - [Grand Theft Auto](#) (1997)
- [perspective cameras and 3D worlds]
  - [Night Driver](#) (1976)
  - [Elite](#) (1984)
  - [No Man's Sky](#) (updated during 2021)

# Cameras in animated films

[old] [Walt Disney's MultiPlane Camera](#) (1957)

[new] [Why 'The Mandalorian' Uses Virtual Sets Over Green Screen](#) (2020)

We're not done figuring out how to map fictional worlds onto 2D display devices:

[redirected walking in VR](#)

# Do we ever need to use multiple cameras?

---

# Multiple cameras for a single videogame display



Need for Speed (1994)

# Phaser's cameras

- **Cameras as images:** background color, position, size, ...
- **Cameras as objects in the world:** rotate, zoom, follow, lerp, ...

Reference guide:

<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/camera/>

```
// default camera
scene.cameras.mains

// make more cameras!
scene.cameras.add(x, y, width, height)
```

Let's just look at an example of four views on a single world with some cool camera effects:  
<https://phaser.io/examples/v3/view/camera/camera-effects>

# Cameras usually need to be controlled

(to keep focus where you want it and to avoid embarrassment)

Keeping the focus (*following, deadzones, bounds*, etc.):

[https://www.gamasutra.com/blogs/ItayKeren/20150511/243083/Scroll\\_Back\\_The\\_Theory\\_and\\_Practice\\_of\\_Cameras\\_in\\_SideScrollers.php](https://www.gamasutra.com/blogs/ItayKeren/20150511/243083/Scroll_Back_The_Theory_and_Practice_of_Cameras_in_SideScrollers.php)

Let's look at a camera control bug in one of Adam's old projects (for CMPM 147): <https://glitch.com/~one-forty-seven>

# Cameras in action

Let's look at how Phaser's cameras are used in Nathan's [CameraLucida](#) example.

# Break

---

# Assets

---

# Art Data Assets

- Some benefits:
  - Artists can shape the player experience without touching (much) code!
  - ...
  - Wait, what's the alternative? [Procedural art](#).
- Design considerations:
  - Can I delay loading assets that might not be seen? Most people only see the first few minutes of a game anyway.
  - Can I make my asset files smaller (in bytes) without hurting quality much?
  - Can I bundle assets together for efficient loading?
  - ...
  - How hard is it for artists to change a single pixel and see the result in-game?
- Some drawbacks:
  - Loading time breaks immersion and flow.
  - Energy usage while loading discourages mobile play.
  - Large downloads may cause players to hit

# Aside: Mobile energy (battery usage) usage



**Radio communication** (downloading assets; multiplayer data)  
**Processor usage** (showing distracting animations; running game)

**Screen brightness** (a white LED flashlight with a colored LCD in front)

Great mobile experience for podcast apps:

- Data is loaded in bulk (not many tiny segments) and then radio can go quiet.
- Data is saved between sessions (no re-downloading).
- Start listening right away (while download continues).
- Screen goes dark.
- CPU wakes up only rarely to decode a segment of audio before going right back to sleep.

Poor mobile experience for in-browser games:

- Data loaded in many chunks, each involving separate network transactions; radio active for a long time.
- Data rarely cached between sessions.
- Player watches animation on loading screen with almost full brightness and CPU usage.
- Screen never goes dark while playing.

You're battery is dead in an hour! (Installable apps improve the experience somewhat but they don't address processor/screen

# In the olden days...



Game cartridges for consoles used to contain (still do!) memory chips.

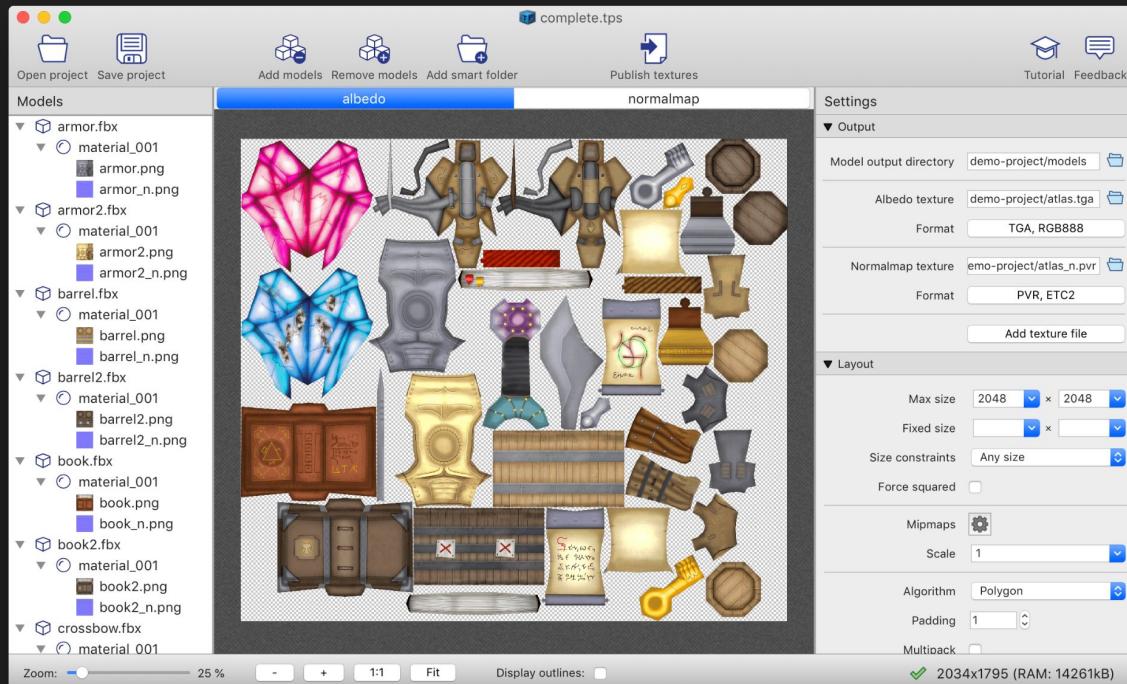
Pros:

- Very fast load time!

Cons:

- Expensive to manufacture/ship
- Player must wait for physical stock in store to buy
- Tedious for artists to update assets during development

# In the present day... (assuming you aren't making custom carts)



<https://www.codeandweb.com/texturepacker3d>

We have flexibility about loading many small individual assets or bundling them together for efficient loading using advanced tools. (Using “texture atlases” and other bundles.)

## Pros:

- More efficient use of radio+processor
- Better memory use (wider audience with older devices)

Major con: Artists must re-run the bundling tool to produce the files used by the game.

Metaphor: Is it faster to walk, bike, drive, train, plane? It depends on the destination and frequency of trip.

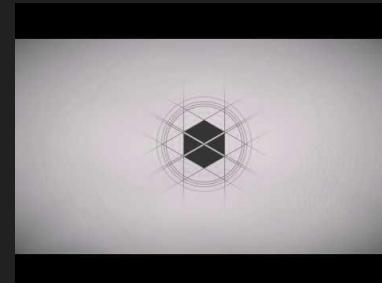
# Hiding your loading time



Sometimes you can hide the loading time by making your loading screens feel a little more like the gameplay screens, but most players will see right through this.

Trouble: When do you load the assets that are used to play the loading screen???

Sometimes simple procedural graphics are best.



# Assets in the *Phaser 3* framework

---

# “Notes of Phaser 3”

The screenshot shows a web browser window with the title bar "Notes of Phaser 3". The address bar contains the URL "rexrainbow.github.io/phaser3-rex-notes/docs/site/index.html". The page itself has a blue header bar with the title "Notes of Phaser 3" and a search bar. On the left, there's a sidebar with a navigation menu for "Notes of Phaser 3" containing various game development components like Phaser, Loader, Game object, Input, Audio, Camera, Tween, Shader effects, Logic, Data, Math, String, Time, UI, Board, and Firephaser. The main content area is divided into sections: "Home" (with a "Phaser" section), "Table of contents" (listing Phaser, Phaser3, and Rex plugins), "Links" (with a "Phaser3" section containing links to FAQ, Official discord channel, API document, Examples, and Bug report).

An excellent and fast-loading reference for Phaser 3: (bookmark it)  
<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/index.html>

# A few of Phaser's loadable data types

- Images (good for static images)
- Sprites (moving, animated game objects)
- Sprite Sheets (a grid of tiled images)
- Texture Atlases (flexible collection of named sub-images)
- ...
- Audio (single audio file)
- Audio Sprites (a collection of name sub-clips)  
<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/loader/#file-types>
- ...



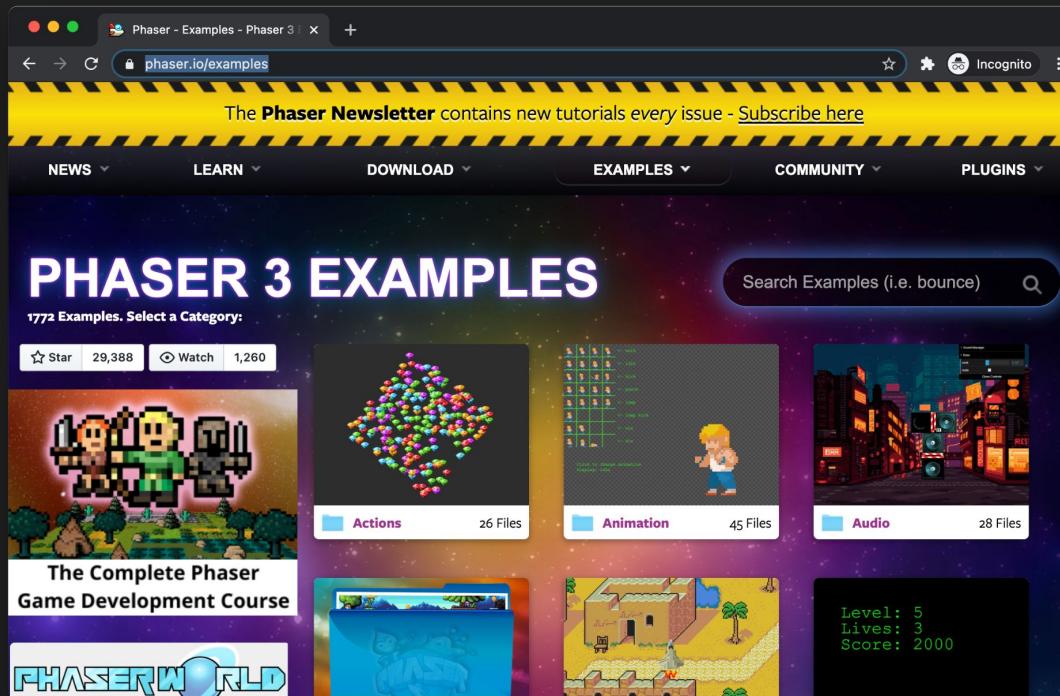
# General asset usage pattern

```
class MyScene extends Phaser.Scene {  
  
    preload() {  
  
        this.load.image("ship", "assets/ship.png");  
  
    }  
  
    create() {  
  
        this.myShip = this.add.sprite(10, 20, "ship");  
  
    }  
}
```

file path on your computer

key / nickname (don't need to match filename)

# Phaser's searchable example index



Browse: <https://phaser.io/examples>

Source: <https://github.com/photonstorm/phaser3-examples/tree/master/public>

# Nathan's example projects for CMPM 120 specifically

<https://github.com/nathanaltice?tab=repositories>

Let's clone and inspect **AssetBonanza**.

Plan:

- Let's look at where the assets are being loaded in the Network devtools page.
- Let's display many many more assets and look at effects on FPS meter.

# What can I do with a game object (e.g. Sprite, etc.)?

<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/gameobject/>

Destroy, Position, Angle, Origin, Size, Click, State, Add to Group, etc.

# Advanced questions

- Can I show stuff on the screen while assets are loading? (like in *Destiny 2*)
  - <https://phaser.io/examples/v3/view/loader/loader-events/load-progress>
- Can I use an asset on the loading screen? (like in *Mass Effect* elevators)
  - <https://phaser.io/examples/v3/view/loader/loader-events/play-animation-during-load>
- Can I start play before the assets are loaded and have them pop in when ready? (like *Mass Effect* core gameplay)
  - <https://phaser.io/examples/v3/view/loader/loader-events/display-file-as-loaded>

## If time permits...

- Let's check out the Leshy SpriteSheet tool for making texture atlases:  
<https://www.leshylabs.com/apps/sstool/>
- Let's talk about procedural graphics in Phaser:  
<https://phaser.io/examples/v3/view/camera/graphics-landscape>

# Break

---

# Scene Transitions

---

# Creating meaning through editing

Kuleshov effect: how you cut between scenes influence how the audience interprets the relationship between what is literally shown in each scene.

Phaser's scenes:

- Does it feel like one scene is contained within or stacked above another?
- Does it feel like we've returned to a scene that continued to exist or are we visiting a fresh instance of the scene for the first time?

# Phaser's Scene states

States: running, paused, sleeping,  
(nonexistent)

Actions:

- restart
- pause/resume
- wake/sleep
- switch (sleep one, wake the next)
- start (destroy one, launch the next)
- transition(config) (let both coexist for a controlled period)

Stacking:

- Default display order comes from game config object
- `this.scene.moveAbove(keyA, keyB)` to customize

|       | Update/isActive | Render/isVisible |
|-------|-----------------|------------------|
| Run   | ✓               | ✓                |
| Pause |                 | ✓                |
| Sleep |                 |                  |

<https://rexrainbow.github.io/phaser3-rex-notes/docs/site/scenemanager/>

# One scene at a time

Title scene

Play scene

Credits scene

Pause scene

# Pause/resume

Play scene

Pause scene

To keep the action frozen during pause:

On pause key:

```
this.scene.pause();  
this.scene.launch('pause');
```

On resume key:

```
this.scene.stop()  
this.scene.resume('play')
```

# Sleep/Wake

Title scene Play scene

To not replay an exciting title-intro animation each time:

On play key:

```
this.scene.sleep();  
this.scene.launch('play');
```

On play session over:

```
this.scene.stop()  
this.scene.wake('title')
```

# Real-time inventory management

Play scene

Inventory scene

To keep gameplay alive while in a sub-scene:

On inventory open:

```
this.scene.launch('inventory', data);
```

On inventory close:

```
this.scene.stop()
```

# Nathan's Scenesters project

Let's briefly check out how the scene transitions happen.

# Bonus Slides

---

# Exit Slip

---

<https://forms.gle/rZJBZTRrgAZUhJnWz>