Prefabs and Prototypes
Reviewing the First Assignment
The best time to turn it in is before the deadline

The second best time is now
Phaser in separate folder

framework

index.html

projectName

main.js

styles.js

audio

img

Rename this!
"All long help threads should have a sticky globally-editable post at the top saying 'DEAR PEOPLE FROM THE FUTURE: Here's what we've figured out so far ...'"

https://xkcd.com/979/
Please help each other!

Getting help from other people is **good**...

...**just cite** where the code or ideas came from.

...it can also help if you type the code in yourself instead of just pasting it.

This is a good idea because it's important to understand what your code is doing.

Code on Stack Overflow can be **wrong**
Why You Are Here

➔ Learn the **basic principles** of game programming and put them into practice

➔ Learn how to do the low-level implementation so we can turn **ideas** into **working games**

➔ Learn how **technology** *and* **teamwork** affect game design (PLO 7 & 8)

**Note:** Not (directly) testing you on knowing Computer Science concepts! That's what classes like 12B are for!
Document your process

If your workflow involves following a particular set of steps, write that down.

This applies to artists too!
Comments

// Slight change in how comments will be graded going forward

// You're all good at telling me the **how**

// But I also want to know the **why**

// If how something works seems obvious to you, less need for comments -- though it may not be obvious to the rest of your team.
Literate Programming

// The original literate programming paper

Objects & Prefabs
Learning Objectives

By the end of class you should be able to...

➔ Paraphrase what a game prefab is
➔ Explain how to use JavaScript objects to...
 ◆ ...extend an existing prototype
 ◆ ...make prefabs and organize your code using prototypes

Homework Assignment #2

➔ Demonstrate organizing a game's files
➔ Implement multiple game states
➔ Practice making your code modular
What are some reasons to have more than one file for our games?

- Too much information
- Logical organization
- C++ re-compiles faster
- teamwork!
Game Prefabs

Games are complicated systems, we need organization.

One way this is done is called a prefab.

A prefabricated object - same code and data gets used many times

➔ In Phaser, prefabs are usually:
  ◆ in-game elements that extend Phaser.Sprite
  ◆ or related user interface elements that extend Phaser.Group.
A prefab will add properties and methods that make the extended Phaser object unique
Inherited properties and methods

Prefab properties and methods

this.jumpHeight
this.runSpeed
etc.
If you have experience with other programming languages

JavaScript does things a bit differently
You are probably familiar with grouping code into functions for organization and reuse.

function addFive(parameter) {
    return parameter + 5;
}

In Javascript, functions are a type of object.

return addFive;
// Type 1: Function declarations
// no parameters or return
function useless() {
  console.log('nihilism');
}

useless();   // :(

// Note: primitive parameters (like number) are passed *by value*

function cube(num) {
  return num * num * num;
}

cube(10);   // 1000
cube(8);    // 512
cube('3');  // ???
cube('cat'); // NaN

// Type 2: Function expressions
// Really handy for passing a function as an argument to another function

// anonymous style
var cubed = function(num) { return num * num * num; }

var x = cubed(4);   // x = 64

// or named (useful for function recursion)
var exp = function power(base, exponent) {
  console.log('exponent: ' + exponent);   // see the recursion happen
  if(exponent === 0) return 1;
  else return base * power(base, exponent - 1);
}

var a = exp(9, 0);   // a = 1
var b = exp(8, 3);   // b = 512
Functions inside functions

```javascript
function addANumber(a_number) {
    var adder = function(parameter) {
        return parameter + a_number;
    }
    return adder;
}

var add_five = addANumber(5);
add_five(10);
```

By the way, this is called a closure.
Variable bindings are only valid in part of the program.

This region is called the scope.
let versus var

function exampleFunctionOne() {
    let first = 7;
    console.log(first);
    for(let first = 0; first < 5; first++) {
        console.log(first);
    }
    console.log(first);
}

The `let` statement declares an enclosing block scope local variable.

function exampleFunctionTwo() {
    // hoisting: var second;
    console.log(second);
    for(var second = 0; second < 5; second++) {
        console.log(second);
    }
    console.log(second);
}

The `var` statement declares a function scope variable.
Lexical Scope versus Closures

function parent() {
    var parent_value = 1;
    function child() {
        var child_value = 2;
    }
}

function makeAdder(x) {
    return function(y) {
        return x + y;
    }
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(2));  // 7
console.log(add10(2)); // 12

Lexical scope exists in the written code: the parent_value is accessible in the child function, but the child_value isn't accessible in the parent function.

Closures use the run-time context from when the outer function was called and the inner function was created.
Functions are objects, objects have properties

Therefore, functions can have properties.

You'll remember this later.
Arrays

An array is an ordered set of objects that you can access by index.

```javascript
// define an array literal (preferred method)
var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];

// other valid array definitions
var fibonacci = new Array(0, 1, 1, 2, 3, 5, 8, 13, 21, 34);
var tacoBellSauces = Array('mild', 'hot', 'fire');

// arrays can hold mixed types (including other objects)
var nonsense = [11, 'tacos', 8.88, null, [3, 8], true, {life: 0, death: 1}];

// an array's first index is [0]
var x = primeNumbers[0]; // x = 2
var y = tacoBellSauces[3]; // y = undefined
var z = nonsense[1]; // z = 'tacos'

// arrays are special objects that have some built-in properties...
var len = primeNumbers.length; // len = 10

// ...and some of those properties are methods
primeNumbers.push(31, 37, 41, 43); // add values to the *end* of the array
var popped = primeNumbers.pop(); // removes/returns last (popped = 43)
var shifted = primeNumbers.shift(); // removes/returns first (shifted = 2)
var fibList = fibonacci.join('|'); // fibList = "0|1|1|2|3|5|8|13|21|34"
var revFib = fibonacci.reverse(); // revFib = [34, 21, 13, 8, 5, 3, 2, 1, 1, 0]

// There are lots of other handy array methods!
```
Looping through arrays

```javascript
var ants_of_california = ["argentine ants", "forelius pruinosus", "bicolored pyramid ant", "odorous house ant", "ghost ant", "velvety tree ant"];
for(let i = 0; i < ants_of_california.length; i++) {
    console.log(ants_of_california[i]);
}

ants_of_california.push("argentine ants");

for(let ant of ants_in_california) {
    console.log(ant);
}
```
Objects

Most things in JavaScript are objects

Objects are arbitrary collections of properties

Properties that are bound to functions are called methods

We can access, reassign, and enumerate an object's properties
We can use objects to organize things in our game.

Using the object values instead of writing them explicitly in `preload()`
Properties can have functions: we call this a method
What if we want multiple, slightly different objects?

```javascript
// What if we want multiple objects that are slightly different?
var creature1 = {
  image: "1F42C.png",
  namekey: "dolphin",
  velocity: 200,
  vertical: 50,
  bounce: 1.0,
  switch_direction: function() {
    creature.velocity *= -1;
    console.log("switched directions");
  }
};

var creature2 = {
  image: "1F40D.png",
  namekey: "snake",
  velocity: 200,
  vertical: 250,
  bounce: 1.0,
  switch_direction: function() {
    creature.velocity *= -1;
    console.log("switched directions");
  }
};

// This, as you might guess, doesn't scale.
```
Share setup by using a constructor function!

Note the use of the `new` keyword

Where have we seen the `new` keyword before?

We can add methods to existing objects!
Phaser.Game() is a constructor function!

You've already used constructor functions!
new & this

Calling a function with the `new` keyword causes it to be treated as a constructor.

The constructor will have its `this` variable bound to a fresh object.
this refers to the object the code is inside
Q: Will this one-line program throw a browser error?

```javascript
this.greeting = "Hello World";
```
Prototypes
Prototypes

You might have noticed something unexplained last week:

In our Phaser states example what is MainMenu.prototype?
“Every JavaScript object has a second JavaScript object associated with it. This second object is known as a prototype, and the first object inherits properties from the prototype.”

Two prototype concepts

Every JavaScript object has a prototype attribute that points to its “parent,” i.e., the object from which it inherited its properties. This attribute is normally referred to as the prototype object.

The prototype object is a property of each instance.

Every JavaScript function has a prototype property that is empty by default. You implement inheritance by attaching properties and methods to this property.

The prototype property is a property of the constructor.

Object.getPrototypeOf(new ConstructMyObject()) == ConstructMyObject.prototype;
Object literals all have the same prototype object

All objects created with {} have `Object.prototype` as their prototype object

```javascript
var playerSprite = {
  x: 200,
  y: 200,
  src: "dolphin.png"
}
```
Editing the Constructor

We know how to add new properties to existing objects.

But what if we want to add a shared property to the constructor itself?
function Creature(key, image, velocity, vertical) {
    this.image = image;
    this.namekey = key;
    this.velocity = velocity;
    this.vertical = vertical;
    this.bounce = 1.0;
}

Creature.prototype = {
    switch_direction: function() {
        this.velocity *= -1;
        console.log("switched directions");
    },
    move: function() {
        this["acceleration"] = 5;
    };
};

We add to the prototype with .prototype

Creature

new

move()

swim()

move()

slime_trail()
You can think of it like how some cards in Magic: the Gathering alter how cards of a particular type work.

Image credit: @roborosewater
https://twitter.com/RoboRosewater/status/972577767576489984
.prototype and Phaser

We use prototypes with our Phaser states.

How does it work?
Adding States

Tell Phaser we want to add an object that enacts a state, with the key of 'MainMenu'.

Say that we want to start with the state identified with the key of 'MainMenu'.

```javascript
// add states to StateManager and start MainMenu
game.state.add('MainMenu', MainMenu);

game.state.add('GamePlay', GamePlay);

game.state.add('GameOver', GameOver);

game.state.start('MainMenu');
```
Phaser looks up the 'MainMenu' key and sees that it points to a function called MainMenu()

Note that the game object has been passed to the function so we can reference it within the state.
Adding States

Normally, these would be adding new properties

But because we added MainMenu to Phaser's StateManager, Phaser made MainMenu an instance of its State object.
Adding States

So now these are overriding inherited properties
For **State** methods that you don't override, JavaScript moves up the **prototype chain** until it finds them.

Phaser defines all of the **State** methods you see above, but *none of them have any default behaviors*—it's up to you to provide them. 😊
Gloom

If it helps, you can think of this like the transparent cards in the game Gloom, where cards on top override the cards underneath.

Image credit: Atlas Games
https://www.atlas-games.com/gloom/
Constructing a prefab

Step by step
function Player(game, key, frame, scale, rotation) {
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
  game.rnd.integerInRange(64, game.height-61), key, frame);

  // add properties
  this.anchor.set(0.5);
  this.scale.x = scale;
  this.scale.y = scale;
  this.rotation = rotation;

  game.physics.enable(this);
  this.body.collideWorldBounds = true;
  this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
    this.body.angularVelocity += 5;
  }
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
    this.body.angularVelocity -= 5;
  }
}
function Player(game, key, frame, scale, rotation) {
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width - 64),
    game.rnd.integerInRange(64, game.height - 61), key, frame);

    // add properties
    this.anchor.set(0.5);
    this.scale.x = scale;
    this.scale.y = scale;
    this.rotation = rotation;

    game.physics.enable(this);
    this.body.collideWorldBounds = true;
    this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
    if (game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
        this.body.angularVelocity += 5;
    }
    if (game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
        this.body.angularVelocity -= 5;
    }
}
function Player(game, key, frame, scale, rotation) {
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
                          game.rnd.integerInRange(64, game.height-61), key, frame);

    // add properties
    this.anchor.set(0.5);
    this.scale.x = scale;
    this.scale.y = scale;
    this.rotation = rotation;

    game.physics.enable(this);
    this.body.collideWorldBounds = true;
    this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
    if (game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
        this.body.angularVelocity += 5;
    }
    if (game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
        this.body.angularVelocity -= 5;
    }
}
call()

“JavaScript functions are objects and like all JavaScript objects, they have methods.”

call() allows you to indirectly invoke a function as if it were a method of some other object. The first argument is the object on which the function is to be invoked; this argument becomes the value of the this keyword within the body of the function.”

“Any arguments to call() after the first invocation context argument are the values that are passed to the function that is invoked.”

function Player(game, key, frame, scale, rotation) {
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
    game.rnd.integerInRange(64, game.height-64), key, frame);

    // add properties
    this.anchor.set(0.5);
    this.scale.x = scale;
    this.scale.y = scale;
    this.rotation = rotation;

    game.physics.enable(this);
    this.body.collideWorldBounds = true;
    this.body.angularVelocity = Phaser.InGame.physics.arcade.defaultAngularVel;
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
    if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
        this.body.angularVelocity += 5;
    }
    if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
        this.body.angularVelocity -= 5;
    }
}
Player

Inherited properties and methods
- this.jumpHeight
- this.runSpeed
- etc.

Prefab properties and methods

Phaser.Sprite

First argument this - i.e. the Player object

Other arguments Player.Sprite() parameters

Public Properties
- alive
- anchor
- add
- animations
- autoCall
- blendMode
- body
- bottom
- bounds
- children
- components
- cropRect
- data
- destroyPhase
- events
- exists
- excluded
- exists
- freezeFor
- heal
- health
- height
- hidden
- isWorld
- key
- left
- lifespan
- exists
- outOfGamersBoundsKill
- previousRotation
- renderOrder
- sma
- smoothed
- texture
- tint
- tintedTexture
- top
- transformCallback
- transformCallbackContext
- type
- width

Public Methods
- addChild
- addChildAt
- align
- alignTo
- alignX
- bringToFront
- contains
- crop
- destroy
- getBounds
- getChild
- getChildIndex
- getChildrenBounds
- kill
- loadTexture
- moveDown
- moveUp
- overlap
- play
- postUpdate
- preUpdate
- removeChild
- removeChildAt
- removeChildren
- reset
- resetFrame
- resizeFrame
- revive
- sendToBack
- setChildIndex
- setFrame
- setScale
- setTexture
- swapChildren
- update
- updateCrop
Now we can extend the default Phaser.Sprite by adding our own properties!

```javascript
function Player(game, key, frame, scale, rotation) {
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width - 64),
                         game.rnd.integerInRange(64, game.height - 61), key, frame);

    // add properties
    this.anchor.set(0.5);
    this.scale.x = scale;
    this.scale.y = scale;
    this.rotation = rotation;

    game.physics.enable(this);
    this.body.collideWorldBounds = true;
    this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
    if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
        this.body.angularVelocity += 5;
    }
    if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
        this.body.angularVelocity -= 5;
    }
}
```

Note the use of the `this` keyword to refer to our own object!
function Player(game, key, frame, scale, rotation) {
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
                        game.rnd.integerInRange(64, game.height-64), key, frame);

    // add properties
    this.anchor.set(0.5);
    this.scale.x = scale;
    this.scale.y = scale;
    this.rotation = rotation;

    game.physics.enable(this);
    this.body.collideWorldBounds = true;
    this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
    if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
        this.body.angularVelocity += 5;
    }
    if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
        this.body.angularVelocity -= 5;
    }
}
function Player(game, key, frame, scale, rotation) {
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
                        game.rnd.integerInRange(64, game.height-64), key, frame);

    // add properties
    this.anchor.set(0.5);
    this.scale.x = scale;
    this.scale.y = scale;
    this.rotation = rotation;

    game.physics.enable(this);
    this.body.collideWorldBounds = true;
    this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
    if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
        this.body.angularVelocity += 5;
    }
    if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
        this.body.angularVelocity -= 5;
    }
}
Back in main.js, we use our Prefab constructor to create three new Player objects.
Note that we have to manually add our prefab objects to Phaser's display list.
How the parameters flow

```javascript
player = new Player(game, 'atlas', 'diamond', 3, 45);

function Player(game, key, frame, scale, rotation) {
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
  game.rnd.integerInRange(64, game.height-61), key, frame);

  this.scale.x = scale;
  this.scale.y = scale;
```
Need to include the `<script>` file in `index.html`

The order is important!
Press up/down to change angular velocity

inheritance02.js / Player.js
Project organization

Keeps things manageable

Helps with cooperation
Programming Homework #2
Snowy States
Snowy States

→ Organization
  ◆ Comments
  ◆ File Structure

→ States and Conditions
  ◆ Your game should have three states: MainMenu, Play, and GameOver
  ◆ Use the state object's .prototype
  ◆ Add text and additional behaviors, as described in the assignment

→ Prefabs
  ◆ Construct a Snowstorm prefab in a separate file
  ◆ Add 100 snowflake objects to the scene
  ◆ Override the prefab's update method to allow the player to reverse all of the snowflakes