Object-Oriented Programming

Last Time

Interactivity!

External signals (keypresses, mouse movement, clicks, etc.) are turned into *events* by software systems.

Several functions within Processing are automatically called if they exist, when the appropriate event occurs.

- `mousePressed()`
- `keyPressed()`
- `keyReleased()`
- `mouseMoved()`

Questions

This event logic is cute and all, but what would real programs that handle external events look like?

From the Minecraft Forge custom server project:

```
1 public class ListenerClass {
2     @SubscribeEvent
3     public void onPlayerLogin(PlayerLoggedInEvent event) {
4         event.player.addChatMessage(
5             new TextComponentString("Welcome to the server!")
6         );
7     }
8 }
```

```
1 MinecraftForge.EVENT_BUS.register(new ListenerClass());
```

How do we use `loop()`, `noloop()`, and `redraw()`?

Two general patterns that are good when you're first starting out:

1. Just use the default of drawing every 60 seconds.
2. Call `noloop()` in `setup()` and then call `redraw()` every time the screen needs to change.

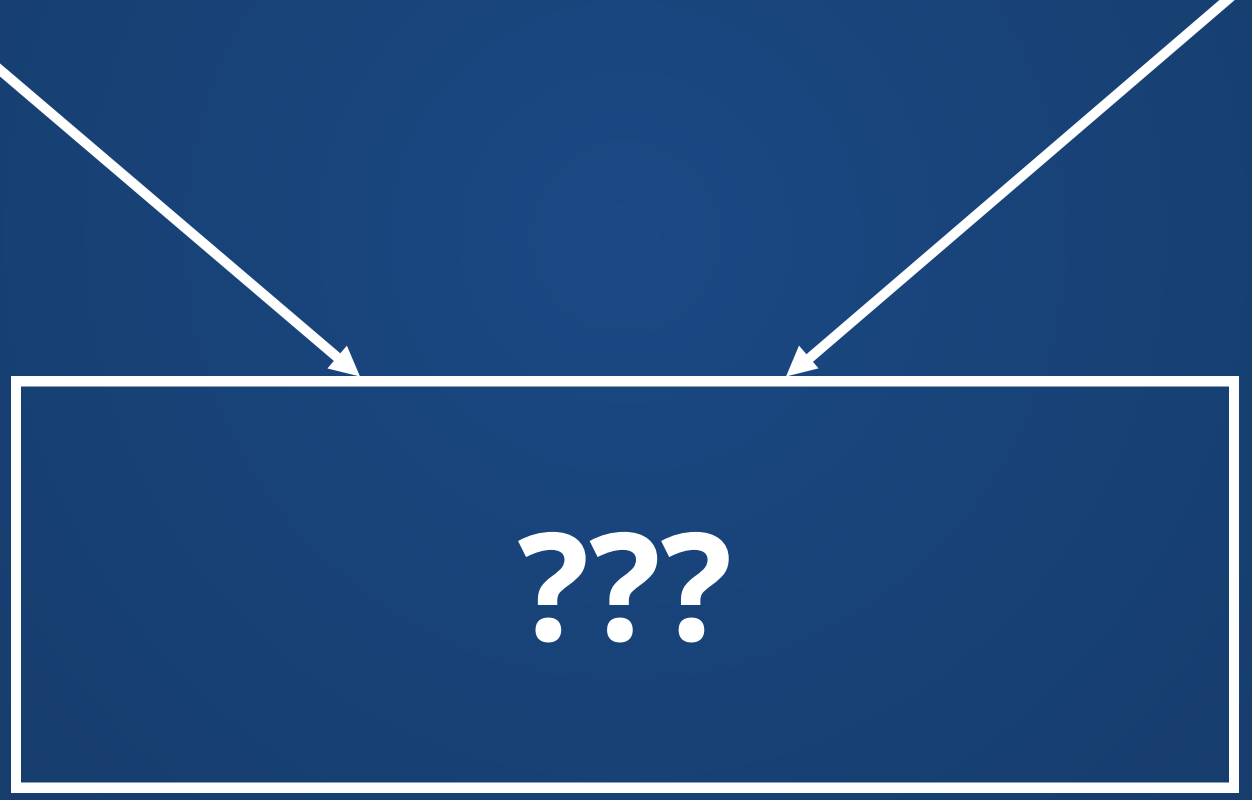
Do <u>not</u> try to use these functions to make your animation behave correctly, because it's very likely to fail once things get complex!

Element 1

Needs noLoop() to be called when it appears and redraw() to be called when it disappears.

Element 2

Needs the draw-loop to be running for as long as this element is visible on the screen.



Why would we want to put `mousePressed` inside of draw instead of just calling it?

There are actually (at least) two things called `mousePressed` in Processing:

- A *function* which is called every time the mouse is pressed.
- A *variable* which is set to true when the mouse has been pressed (and false once it's been released).

Don't call your event handlers manually--- Processing knows how to call them when the appropriate event occurs.

Why can't Processing accept non-ASCII input or more than one input at a time?

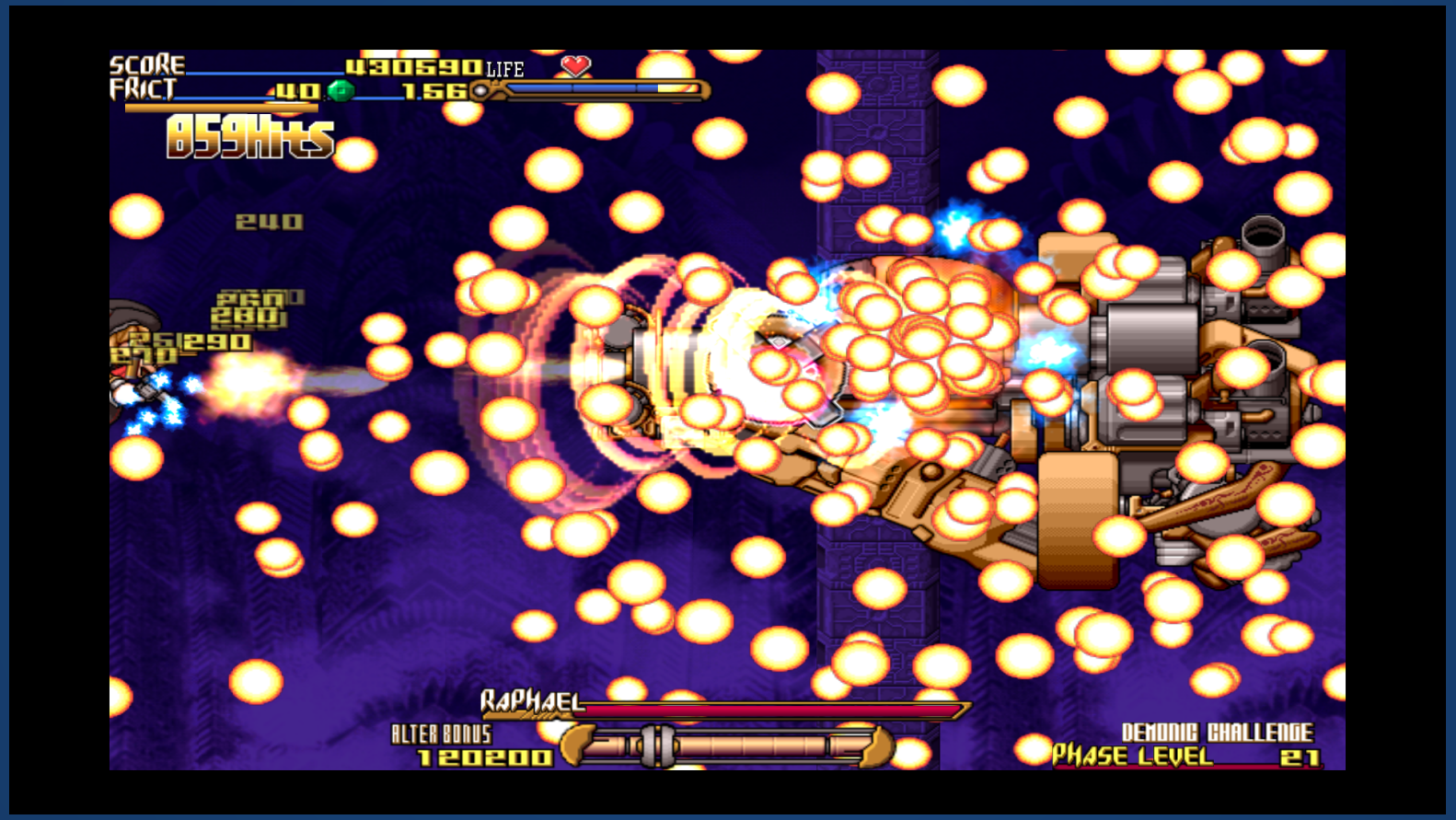
More than one input at a time: you can handle this yourself.

We have:

- A function which triggers when a single key is pressed.
- A function which triggers when a single key is released.
- A variable which stores the most recent key manipulated (either pressed or released)
- The ability to create global variables.

How do we detect multiple keys being held?

Consider a bullet-hell game



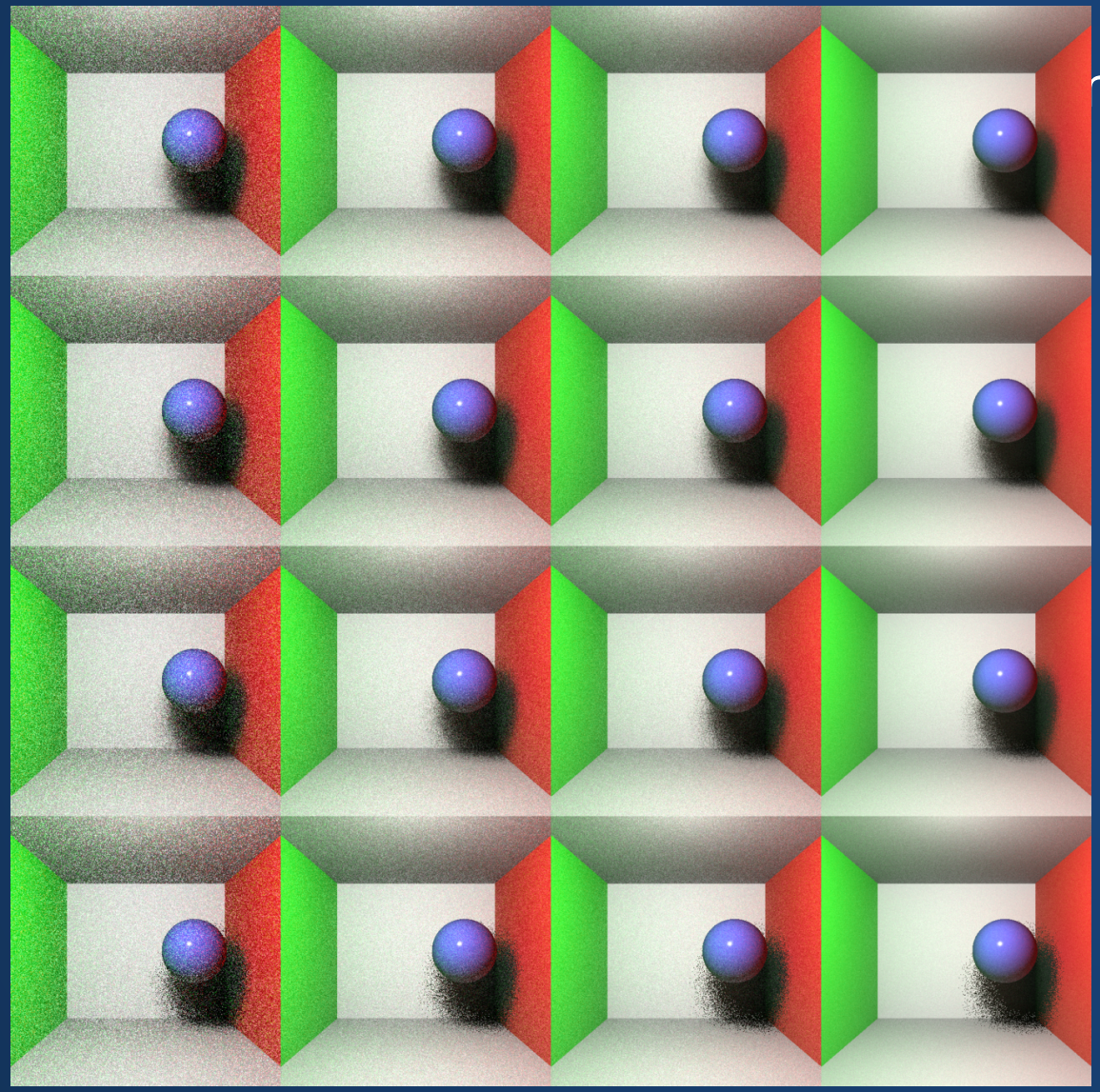
It seems like drawing bullets, checking collisions, and moving a player object might be a lot for a CPU to handle.

Why are the shapes that I draw flickering in and out of existence?

Simplifying Code

So far, all the systems we've worked on have been relatively simple.

Real graphics are complex!



nples/multipleparticlesystems.html

- Codebase size:
  - 3000 lines of C
  - 8500 lines of C++ code
  - 17,000 lines of C++ headers
  - Doesn't count external libraries for doing math, handling vectors, etc.

How can we control the complexity of our code?

Alt. Phrasing: How can we make it harder to get things wrong?

Case Study: A Car

We're going to model a very simple car.

Our car has two variables: fuel and speed.

It can do three things:

- Accelerate: reduce fuel and add speed
- Decelerate: reduce speed
- Refuel: add fuel

What are some logical limits on how these actions work?

We're going to model a very simple car.

Variables:

- Speed
- Fuel

Actions the car can take:

- Accelerate
- Decelerate
- Refuel

Rules that we might want to enforce:

- A car cannot accelerate if it has no fuel.
- A car cannot decelerate if its speed is zero.
- A car can never have negative fuel.
- A car cannot refuel past its tank capacity.

If we accidentally break one of these rules, we've generated a bug!

Enforcing Rules: Attempt 1

Just remember to apply the rules!

```
1 float car1_speed = 0.0;
2 float car1_fuel = 0.0;
3
4 // Lots of intermediate code
5
6 car1_speed += 100.0;
```

Remembering is not safety

Enforcing Rules: Attempt 2

Use functions so that we can't forget to do all the operations.

```
1  // Magic floats which have changes
2  // reflected outside the function
3  void accelerate(float fuel, float speed){
4    if (fuel == 0.0){
5      return;
6    } else if (fuel <= 0.5){
7      // Use what fuel we have left
8      speed += 100.0 * (fuel / 0.5);
9      fuel = 0.0;
10   } else {
11     speed += 100.0;
12     fuel -= 0.5;
13   }
14 }
```

Since this is wrapped in a function, we can afford to be complex now!

Yay!

```
1 float car1_speed = 0.0;
2 float car1_fuel = 0.3;
3
4 accelerate(car1_speed, car1_fuel);
```

```
1 float car1_speed = 0.0;
2 float car1_fuel = 1.0;
3 float car2_speed = 0.0;
4 float car2_fuel = 1.0;
5 // Oops.
6 accelerate(car1_fuel, car2_speed);
```

```
1 float car1_speed = 0.0;
2 float car1_fuel = 0.3;
3
4 // Oops.
5 accelerate(car1_fuel, car1_fuel);
```

...oh. :(

Q: What went wrong?

A: Even though the actions were correct, we were operating on the wrong data.

Enforcing Rules: Attempt 3

Instead of passing the data in separately, we're going to bundle the data together with the functions for manipulating it.

```
1 // float car1_speed = 0.0;
2 // float car1_fuel = 0.3;
3 Car car1 = new Car(0.0, 0.3);
4
5 car1.accelerate();
```

OOP

Organizing Code

Code can get complicated!

Two time-honored tricks for reducing complexity:

- Group related data together
- Worry only about what something does, not how it does it

One of the techniques that evolved out of these two ideas is *Object Oriented Programming*.

OOP: Step 1

Define a *class.*

```
1  class Car {
2    float fuel;
3    float speed;
4
5    Car(float f, float s){
6      // Code goes here
7    }
8
9    void accelerate(){
10     // Code goes here
11   }
12
13 }
```

Note that we don't define values for `fuel` and `speed`. This is because the class acts as a **blueprint** for instances of `Cars`.

Elements of this class (`fuel`, `speed`, `accelerate()`) are known as *members*.

Data members are known as *fields.*

Function members are known as *methods*.

OOP: Step 2

Create objects from the class and use them.

```
1 Car c1 = new Car(0.0, 0.0);
2 Car c2 = new Car(0.0, 0.5);
3 println(c1.fuel);
4 c1.accelerate();
5 println(c1.fuel);
```

Things to note:

- The type of the object is the class name (e.g. the type of `c1` is `Car`).
- We need to use the `new` keyword to create an object. This is different from Python!
- We access members of the object using dot-notation.

Class vs Object

Car class

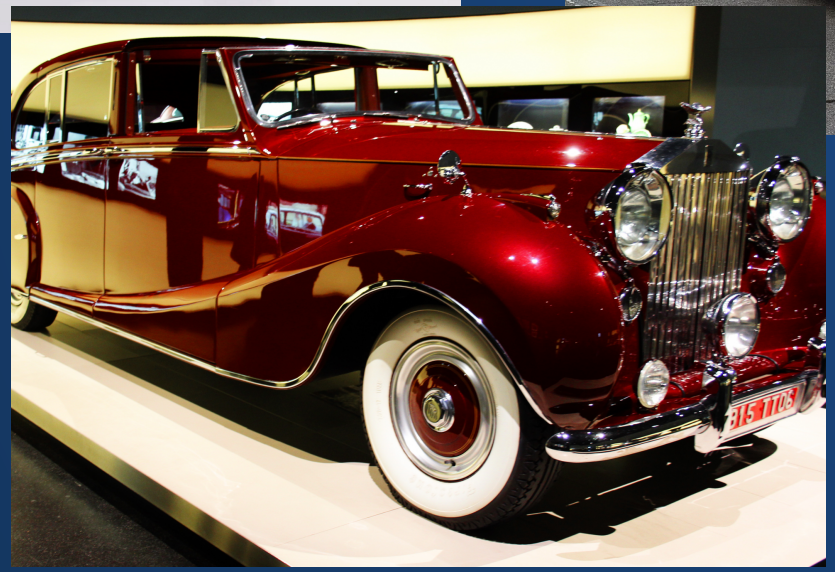
- Name: Car
- Fields:
  - make
  - model
  - color
  - speed
  - fuel
- methods:
  - accelerate(rate)
  - brake(rate)

Car object

- Name: car_7
- Fields:
  - make = "Honda"
  - model = "Civic"
  - color = PURPLE
  - speed = 0
  - fuel = 10.0

A Note on Language Usage

None of these are `Cars`!



Example Class: Spot

```
1  class Spot {
2    float x, y, radius;
3
4    void display() {
5      ellipse(x, y, radius, radius);
6    }
7
8    Spot(){
9      x = 50.0;
10     y = 50.0;
11     radius = 30.0;
12   }
13
14   Spot(float x, float y, float r) {
15     this.x = x;
16     this.y = y;
17     this.radius = r;
18   }
19 }
```

Things to note:

- Constructor is defined as a function within the class, with the same name as the class.
- We can have multiple constructors, as long as they differ in number and type of arguments.
- If not ambiguous, can just use class variable names to refer to object variables.
- Can use `this` keyword to refer to the current object. Similar to `self` in Python.

Using our Spot

```
1  Spot sp1, sp2;
2
3  void setup(){
4    sp1 = new Spot();
5    sp2 = new Spot(75, 80, 15);
6  }
7
8  void draw(){
9    sp1.display();
10   sp2.display();
11 }
```

Class Files

A single file can contain all of a program’s classes BUT please use separate files for each class for this course when submitting projects (for in-class, you can keep it all in one file if you'd like).

Multiple files provide modularity and make it easier to share/reuse code later if you choose to work in groups for the final project.

Class Files

1. In a sketch folder, create the main program with the setup() and draw() functions.
2. Select "New Tab."
3. Name the file after the class it contains.
4. Copy class files to other sketch folders for reuse.

Note: Each Processing sketch can only have one setup and draw function call

Class Functionality

- Fields represent meaningful object values
  - What might `speed` represent in `Spot`?
  - What might `direction` represent in `Spot`?
- Methods represent meaningful object behaviors
  - How could we use a `move()` method in `Spot`?
  - How might we use a `chameleon()` method in `Spot`? Would we need to add fields to support this?

Reminder: these are sometimes collectively referred to as *class members* or just *members.*

Hands-On: Spot Class

1. Implement the `Spot` class in a Processing sketch. Create it within its own file.
2. Add a `speed` field and a `move()` method so the spot's position can update.
3. Implement a `draw()` method for Spot.
4. Create at least two different `Spot` objects that start out with different positions and speeds, then draw things out. (It is okay if the spots eventually move off the screen, as long as it doesn't occur too quickly).
   HINT: Your main draw() function should be 5 lines of code.

```
1  class Spot{
2    int x;
3    /* Other fields */
4
5    // Constructors
6    Spot(){}
7    Spot(int _x){}
8
9    // Methods
10   void move(/* args */){}
11
12 }
```

Objects in Objects

- Objects can be fields of other objects.
  - Allows for better code reuse and cleaner division between concepts
- Example: `PVector` provides support for vectors.
  - Stores x,y,z values as fields.
  - Provides methods with useful mathematical functionality.

https://processing.org/reference/PVector.html

```
1 class Spot {
2   PVector position;
3   float radius;
4   void display() {
5     ellipse(position.x,
6             position.y,
7             radius, radius);
8   }
9 }
```

Designing Classes

What data goes in fields?

- Data that creates a meaningful representation of the object in question.
- Preferably should be non-redundant, otherwise we can have inconsistent data.

What methods should be implemented?

- Functionality that has a clear purpose and is likely to be called multiple times
- Helper methods are smaller methods that can assist in building out clean functionality

Designing Classes

Unfortunately, design has no hard and fast rules!

- Take problem into consideration before starting the design
- Use naming conventions for both fields and methods that express the purpose of that variable or function
- If possible, avoid writing the same functionality out in multiple places
  - If you find yourself copy-pasting code, ask if that code can be put in a method. The answer won't always be yes, but you should think about it!

Calling `new`

- Calling `new` allocates memory for an object. This allocation of memory can be expensive. Doing this too often may degrade performance.
  - Since the draw() loop runs 60 times per second, need to be especially cautious about using `new` in `draw()`.
- Try to create objects infrequently if you can.
  - Create objects upfront in `setup()` instead of every `draw()`.
  - Create objects based on user input in mouse/key callbacks.
  - Create objects using timers (will be discussed later).
  - If you must call `new` from `draw()`, consider saving objects into global arrays so you can reuse them on the next frame. (Dangerous!)
  - Consider using advanced techniques like object pooling.

Index Cards!

1. Your name and EID.

2. One thing that you learned from class today. You are allowed to say "nothing" if you didn't learn anything.

3. One question you have about something covered in class today. You *may not* respond "nothing".

4. (Optional) Any other comments/questions/thoughts about today's class.