Tic is over!
Tic Feedback

- Don’t fall behind!
Announcements

- Alc I released
  - Sign up for design checks!

- Override codes
  - Coming very soon!

- How to use retries:
  - As a late pass: submit up to 1 week after the deadline, and tell the TAs
  - As a redo: submit another release up to 1 week after getting grades back, and tell the TAs

- Grades should be returned within 2 weeks

- Accommodations & Extensions
  - Email James! james_tompkin@brown.edu

- Design Checks & TA Hours
  - How do they work for everyone? More hours later or earlier?
**Screen vs UIElement**

- **Screens** are NOT **UIElements**
- Similar methods, fundamentally different bits of software
- Players interact with **UIElements** to play the game
- **Screens** store everything about the game, not just things users can interact with
  - Specifically, they contain **GameWorlds**
  - More on that later...
This week: Alc I

- 2 week project
- Puzzle Game
- First checkpoint due next week
Lecture 1

Game Worlds
Game World

MOTIVATION
Games are busy...

- All games involve a number of game objects
  - May be many different types of objects
  - May be many instances of the same type of object
  - May be both of the above
- They exist in their own universe
  - If our entire universe is a game, you’re just an object
- We need to take the burden of organizing these objects off the game code
High Level Representation

The Game Objects

- Small collections of functionality
- Hold their own logic and state
- Don’t know much about the rest of the game

The Game World

- The overarching collection of game objects
- Responsible for global logic and facilitating entity logic
- Represents the boundary between program and game
Without a Game World

Viewport
Map
Unit
Projectiles
GameScreen
Application
FrontEnd
AIUnit
Unit
With a Game World

Viewport
GameScreen
Application
FrontEnd

World
Map
Unit
Bullet
AIUnit
Game Worlds

RESPONSIBILITIES
What does a world do?

- Represents all GameObjects in a single space
- Centralizes object management
  - Maintains list of GameObjects
  - Passes ticks and draws to G
Multi-pass Logic

- Ticking and drawing GameObjects in the wrong order leads to undesirable behavior
  - e.g. drawing background over everything else
- World can selectively update state in order
  - E.g. tick all GameObjects so they update position, *then* check for collisions
  - Should have a way for specifying draw order
  - More on this later!
Event Handling

• Passing events can get cumbersome
• Combined mouse-key events can get complicated
• One solution: store input state in GameWorld
  ○ Array/map of booleans.enums instead of key events (one for each possible key)
  ○ Keep track of mouse position/state
General GameWorld contract

public void tick(long nanosSinceLastTick);
public void draw(GraphicsContext g);
public void onKeyPressed(KeyEvent e);
// more device and event types...
public void onMouseDragged(MouseEvent e);
Game Worlds

QUESTIONS?
Lecture 1
Engine Framework
Engine Framework

GAME OBJECTS
What is a game object?

- Everything your GameWorld holds
- Your background
- Your walls
- Your character
- Your enemies
They aren’t everything

• Your screens aren’t
• Your UI probably isn’t
• Saving/loading isn’t
Hierarchical design

- Consider a simple game with:
  - Trees
  - Skeletons

Diagram:
```
  GameObject
     • Has a sprite
   /       \\     \\     \\
  /       \\     \\     \\
Skeleton
    • Can move
   /       \\     \\     \\
  /       \\     \\     \\
Tree
    • Can grow
```
Hierarchical design

- Now we reach a problem
- Where does the Whomping Willow fit in?
Hierarchical design

- Can make a separate parent class and re-implement moving
- Or add below skeleton and add ability to grow
- Both not ideal
  - Only get worse as the game gets bigger
Solution

- Component-based design
- Everything is a game object, and their properties are built by stacking different components
- Leave game objects dumb
- Let their components do all the heavy lifting

GameObject
Has a list of components
Solution

- **GameObjects** are just lists of behaviors
- The behaviors implement all relevant functionality
- *Composition* over inheritance

**GameObject**
- Has a list of components

**HealthComponent**
- Tracks hitpoints

**PhysicsComponent**
- Updates position each tick

**KeyControlComponent**
- Responds to key presses

**CenterComponent**
- Always centers the viewport around this object

**SpriteComponent**
- Has a sprite
Component-Based Design

- The appearance and logic of each object is defined by its Components
- Making new objects is as easy as adding new Components
**GameObject Contract**

- An object needs to:
  - Add a **Component**
  - Remove a **Component**
  - Access a **Component** (based on some identifier tag)
  - Update its **Components** every game loop
    - Tick “tickable” **Components**
    - Draw “drawable” **Components**
  - Know its position and size in the game world (a **TransformComponent**)

- **Tick**

- **Draw**

- **Component**

- **GameObject**

- **TransformComponent**
TransformComponent

• A TransformComponent stores position and size
• The TransformComponent is special
  ○ All game objects should have one, separate from their component list
  ○ It doesn’t even need to implement the Component interface
  ○ Some game objects might not need/have this info; they can default to 0
GameObject Contract

private List<Component> _components;

public void addComponent(Component c);
public void removeComponent(Component c);

// make sure to use .equals() instead of == to compare Strings
public Component getComponent(String tag);
public TransformComponent getTransform(); // you can also add a setter for this

public void tick(long t);
public void lateTick(); // you don’t need this yet, but include it in your interface
public void draw(GraphicsContext g);
Component Contract

- Needs to respond to ticks and draw events (can be empty)

```java
public void tick(long nanosSinceLastTick);
public void lateTick();
public void draw(GraphicsContext g);

public String getTag();
```
Engine Framework

SYSTEMS
Systems

- Organize shared behavior
  - E.g., not all GameObjects will be drawn
  - GraphicsSystem should only hold GameObjects that have drawable behaviors

- Each System stores a list of relevant GameObjects and calls the relevant method on each of them
  - e.g. TimerSystem calls tick(long nanosSinceLastTick) on GameObjects
Registering Objects

- Your choice of **when** to register GameObjects with Systems
  - On instantiation of the relevant component
  - On addition to the GameWorld
  - Both are fine!

- Your choice of **how** to register GameObjects with Systems
  - Manually, e.g. soundSystem.addObject(obj);
  - Automatically, e.g. have system ask each GameObject if it’s interested
Enforcing Draw Order

- **One option**: have set drawing layers
  - Array of Lists
  - Add objects to the relevant list based on what order they should be drawn in
  - onDraw, go through the array, drawing everything in the list at [0], then [1], etc.
  - Guarantees consistent draw order
Enforcing Draw Order

- **Alternatively**: use a `TreeSet`
  - log time lookup/insertion/removal
  - Predictable order (can set a z-index for drawing)
    - Will need a custom comparator to sort objects by z-index, but it’s only a few lines of code
  - Adding the same `GameObject` twice won’t leave a duplicate
  - To change the z-index of an object, simply remove it from the set and add it again

- **Most of the functionality is already written for you!**
  - Iteration will be easier
  - Addition will be easier
  - Removal will be easier
QUESTIONS?

Engine Framework
Viewports

MOTIVATION
Sometimes screen space is hard

• Theoretically everything can be done in screen space
  ○ E.g. specifying size of objects in pixels, and position in pixel offset
• But some things can be very hard
  ○ E.g. panning
• Games shouldn’t have to worry about how the screen draws them
Game Space vs. Screen Space

- In nearly all games, it makes sense to think of the game as existing in its own “space”
- Introducing the Viewport!
Space Conversions

- **Viewports** should know how to draw objects in game space onto screen space
- **Needs:**
  - Upper left corner in screen space
  - Upper left corner in game space
  - Scale/zoom/pixels per game coordinate
- **Game point to screen:**
  1. Minus upper left in game space
  2. Multiply by scale
  3. Add viewport upper left
- **Screen point to game:**
  - Do the OPPOSITE of the steps in REVERSE

---

**EXAMPLE**

Dot in game space: \((1.5, 2.0)\)

1. Minus \((0.5, 0.8)\) gives us \((1.0, 1.2)\)
2. Multiply by \((100, 100)\) gives us \((100, 120)\)
3. Add \((120, 20)\) gives us \((220, 140)\)

The dot should be drawn at \((220, 140)\) in screen space.

**Note:** if your viewport covers your whole screen, then its upper left is simply \((0, 0)\).
Viewports

IMPLEMENTATION
Implementing Viewports

- **Panning:** being able to move the game space upper left within the Viewport
- **Clipping:** selectively drawing the game world within a specific region of the screen (which allows for multiplayer games with two Viewports)
  - This viewport is clipped to the upper-right of the screen

- For this course, you will only be expected to implement **panning**
  - Simple, naive clipping can be done by drawing solid UIElements on top of the Viewport
**Affine Transforms**

- Useful for handling the scaling and translating from game space to screen space!
- JavaFX’s **Affine** keeps track of geometric transforms for drawing
- Can create an **Affine** that converts from game space to screen space
  - Check out appendTranslation(), appendScale(), and others
- How to use the **Affine** once we have it
  - A **GraphicsContext** instance maintains an internal transform
  - The transform is applied to objects before they are drawn
  - Use getTransform() to get the current transformation to modify
  - Use setTransform(...) to set the modified **Affine**
**Affine Transforms**

- If you ever want to do rotations in your game, you should use **Affine**
  - `appendRotation()`
- Never do any rotation calculations yourself
  - It’s not worth it, we promise
Warning!

- **Viewports** are essential to the rest of the class – every assignment from here on will depend on using your Viewport!
  - Having a functioning Viewport is a primary requirement
  - Design well
  - **Test thoroughly**
  - Don’t put off bugs until later weeks
- The TA staff requires the use of **Affines**
QUESTIONS?
Lecture 1

Tips for Alc I
Zooming

• Need to keep track of a zoom factor
  ○ Separate from window resizing!

• Zooming is multiplicative, not additive
  ○ Rolling mouse wheel should * or / the scale factor

• Need to center zooming on the Viewport center
  ○ Otherwise, zooming will focus on the top-left corner
  ○ If you want to get even fancier, you can zoom in on the current mouse location!
Zooming on Viewport Center

• Need to shift the game space to the top-left so it looks like we’re zooming into the center
  ○ Translate by (-viewportWidth / 2, -viewportHeight / 2)
  ○ Scale by zoom value
  ○ Translate by (viewportWidth / 2, viewportHeight / 2)

• There are multiple ways of doing this (setting more affines, or just adjusting the game world’s top-left position). It’s up to you!
Zooming on Mouse Center

• Make a new Affine
• Calculate new scale based on zoom
• Recalculate upper left corner in game coordinates
  ○ newX = gameMouseX - ((screenMouseX - oldX)/scale)
  ○ newY = gameMouseY - ((screenMouseY - oldY)/scale)
• Apply transformations to Affine like when you first created it
JAVA TIP OF THE WEEK

Tips for Alc I
Generics are cool!

• You’ve used generics before... but have you ever written them?
• It’s as easy as:

```java
public class SimpleContainer<T> {
    private T object;

    public void setObject(T ob) {
        object = ob;
    }

    public T getObject() {
        return object;
    }
}
```
Generics are cool!

- Can use extends to bound the type

```java
public class AnimalHouse<A extends Animal> {
    private A animal;
    public void houseAnimal(A a) { animal = a; }
    public void feedAnimal() { animal.eat(); }
}
```

```java
AnimalHouse<Dog> kennel; // okay
AnimalHouse<Rock> mountain; // compiler error
```
Factories

• Static functions that initialize objects
• Useful for creating standard templates for game objects without subclassing

```java
public static GameObject makeElementFire() {
    GameObject o = new GameObject();
    o.addComponent(new ElementComponent("fire"));
    ...
    return o;
}
```
 Enums

- Comparing strings isn’t fun!
  - Prone to typos
  - Harder to maintain (what if you wanted to change a string representation?)

- Use enums! Stored in their own file.
  - `public enum Element { WATER, EARTH, FIRE, AIR }

- Example from previous slide becomes:
  ```java
  o.addComponent(new ElementComponent(Element.FIRE));
  ```
Tips for Alc I

QUESTIONS?
‘Til Next Week!

- Sign up for design checks! :)

60