Lecture 1

Announcements
Tic is (almost) 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
Changes to Hours

- **Design Checks & TA Hours**
  - How do they work for everyone? In person or remote?

- **New Hours:**
  - **Monday**
    - Hours 5-7 Sal
  - **Tuesday**
    - Remote Design Check 6-7 Alex [4 slots]
    - Remote Design Check 7-9 Carlos [8 slots]
  - **Wednesday**
    - Remote Design Check 5-6 Alex [4 slots]
    - Remote Design Check 6-8 Sal [8 slots]
    - Remote / In Person Hours 7-9 Carlos
  - **Thursday**
    - Remote Hours 6-8 Alex
  - **Friday**
    - Remote Hours 3-6 Alex
Lecture Overview

● Seven mandatory lectures
● Hopefully some optional lectures
  ○ Music / sounds
  ○ Perlin noise
  ○ Some other goodies
● Guest Talk 11/8
  ○ Demiurge Studios
    ■ Local Independent Game Studio
Student Introductions

● Name / Pronouns
● Year / Concentration
● Favorite Game(s) or games you’ve enjoyed recently
● Which game show do you think you’d be really good at
  ○ Family Feud, The Price is Right, Big Brother, American Ninja Warrior, etc.
**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...
Common Questions

- Where does tick get called? Where does draw get called?
  - Called for you in Application but, you are responsible for passing the function calls down

- Does Screen extend Application? Does UIElement extend Screen?
  - No, these should be distinct object in your engine.
This week: Alc I

- 2 week project
- Puzzle Game
- First part due next week
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
Example of Game Objects and Game World

Game Objects

1. NPC - conversations with player
2. Door - send player to inside
3. Player - inventory, moving around, etc.
4. Fence - don’t let player walk into it

Game World

- The outside scene itself
- The inside of the building would be a separate game with its own objects

The relationship between Game worlds and Objects is roughly the same as Screens and UI
Without a Game World

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

Viewport

GameScreen

Application

FrontEnd

World

Map

Unit

Bullet

AIUnit
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 GameObjects
- Potentially manages systems (more on this later)
  - “Potentially” since this will be a design decision on your end
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!
Input Event Handling

- Passing events can get cumbersome, especially when many game objects might need to keep track of input
- Combined mouse-key events can get complicated
- One solution: store input state in GameWorld
  - Store an array/map of booleans/enums instead of key events (one for each possible key)
  - Update this storage when the gameworld receives an input event
  - Will still have to pass events for UI, Screen, and App
General **GameWorld** contract

```java
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);
```
QUESTIONS?
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:
  - Chicken
  - Melon
Hierarchical design

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

- Can make a separate parent class and re-implement moving
- Or add below chicken and add ability to fight
- 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

- **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)
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 *GameObject* will be drawn
  - *GraphicsSystem* should only hold *GameObject* that have drawable behaviors
- **Each System** stores a list of relevant *GameObject* and calls the relevant method on each of them
  - E.g. *TimerSystem* calls `tick(long nanosSinceLastTick)` on *GameObject*
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
Draw Order

Notice how the melon is in front of the house, which is in front of the fencing

- This is achieved through draw order, where we specify the *priority* of each object being drawn
- Background elements have “less” priority than foreground elements
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

Z-index is a priority level that you would give to every object (obj with z=99 will draw before z=4)
Recap of Systems

- Systems are a collection of similar GameObjects
- Systems should be able to add & remove* GameObjects, as well as perform some action on all of it’s objects

* Be careful not to remove GameObjects as you are iterating through your list of GameObjects. If needed, make a Removal Queue that removes objects before or after iteration
Engine without systems (Not ideal)

Engine calls like `tick` and `draw` will be passed down to every object.
Engine with Systems (Much better)

We will go over collisions next week -- this is just to show that systems are another example of engine delegation.
QUESTIONS?
Lecture 1

Viewports
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!
Viewport is a camera for your game

Think of the viewport as a portal, or particular view, into your world

Think of what you’d want to be able to do with a camera (panning, zooming, etc.)
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).
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**
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 \((-\text{viewportWidth} / 2, -\text{viewportHeight} / 2)\)
  - Scale by zoom value
  - Translate by \((\text{viewportWidth} / 2, \text{viewportHeight} / 2)\)

- 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
Tips for Alc I

JAVA TIP OF THE WEEK
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(); }
}

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:
  - `o.addComponent(new ElementComponent(Element.FIRE));`
Tips for Alc I

QUESTIONS?
‘Til Next Week!

- Sign up for design checks! :)