Review #1

Indicating State with Graphics

When your game changes how it will respond to user input, it should indicate the change in state by changing what is displayed.

This is an example of a student's Control menu. When the player presses Enter on an option, a dialog appears indicating that pressing a key does something different than before Enter was pressed. This is a good example of communicating state changes to the user.

Avoiding the god-object Anti-Pattern

- If your gameplay view, gameplay, or gamemodel class has dozens or hundreds of fields you could probably make your life easier.

What you don’t want…

- The example to the right has a Texture2D, a Rectangle, a width, and a
- height for multiple types of game entities.

- If you find yourself with multiple objects that share the same properties,
- consider making a class.

- If these fields are only used as intermediate values to calculate constructor parameters or other one-time calculations, use local variables instead or change the constructor to include the relevant calculations.

```
private Texture2D m_textPlayer;
private Texture2D m_textBgp;
private Texture2D m_textArena;
private Texture2D m_textFlea;
private Texture2D m_textBullet;
private Texture2D m_textHead;
private Texture2D m_textBody;
private Texture2D m_textSpider;
private Texture2D m_textScorpoin;
private Texture2D m_textGameOver;

private Rectangle m_rectPlayer;
private Rectangle m_rectNextPlayer;
private Rectangle m_rectBgp;
private Rectangle m_rectArena;
private Rectangle m_rectLives;
private Rectangle m_rectFlea;
private Rectangle m_rectBullet;
private Rectangle m_rectGameOver;


private int cellSize;
private int arenaWidth;
private int arenaHeight;
private int playerWidth;
private int playerHeight;
private int gridOffsetX;
private int gridOffsetY;
private int bulletHeight;
private int bulletWidth;
private int fleaWidth;
private int fleaHeight;
private int headWidth;
private int headHeight;
private int bodyWidth;
private int bodyHeight;
private int spiderWidth;
private int spiderHeight;
private int scorpionWidth;
private int scorpionHeight;
```

```
/// <summary>
/// Contains a texture, a source rectangle, and size data for a game object.
/// </summary>
6 references
class RenderData
{
    0 references
    public RenderData(Texture2D texture, Rectangle bounds, Vector2 size)
    {
        this.Texture = texture;
        this.Bounds = bounds;
        this.Size = size;
    }

    public Texture2D Texture;
    public Rectangle Bounds;
    public Vector2 Size;
}
```

Condensing declarations into a class makes the code much more readable.

```
RenderData Player;
RenderData Flea;
RenderData Spider;
RenderData Scorpion;
RenderData Bullet;

private Texture2D m_textBgp;
private Texture2D m_textArena;
private Texture2D m_textHead;
private Texture2D m_textBody;
private Texture2D m_textGameOver;

private Rectangle m_rectPlayer;
private Rectangle m_rectNextPlayer;
private Rectangle m_rectBgp;
private Rectangle m_rectArena;
private Rectangle m_rectLives;
private Rectangle m_rectFlea;
private Rectangle m_rectBullet;
private Rectangle m_rectGameOver;

private int cellSize;
private int arenaWidth;
private int arenaHeight;
private int gridOffsetX;
private int gridOffsetY;
private int headWidth;
private int headHeight;
private int bodyWidth;
private int bodyHeight;
```

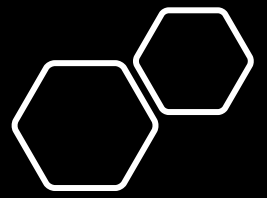
Review #2

```
private List<Objects.Mushroom> m_mushrooms;
private List<Objects.CentipedeSegment> m_segments;
private AnimatedSprite m_headRenderer;
private AnimatedSprite m_segmentRenderer;
private Objects.Player m_player;
private List<Objects.Spider> m_spiders;
private AnimatedSprite m_spiderRenderer;
private List<Objects.Scorpion> m_scorpions;
private AnimatedSprite m_scorpionRenderer;
private List<Objects.Flea> m_fleas;
private AnimatedSprite m_fleaRenderer;
```

```
foreach (var segment in m_segments)
{
    if (segment.m_isHead) m_headRenderer.draw(m_spriteBatch, segment);
    else m_segmentRenderer.draw(m_spriteBatch, segment);
}
foreach (var mushroom in m_mushrooms)
{
    mushroom.draw(m_spriteBatch, m_spriteSheet, new Vector2(64, 0), new Vector2(8, 8));
}
foreach (var flea in m_fleas) m_fleaRenderer.draw(m_spriteBatch, flea);
foreach (var spider in m_spiders) m_spiderRenderer.draw(m_spriteBatch, spider);
foreach (var scorpion in m_scorpions) m_scorpionRenderer.draw(m_spriteBatch, scorpion);
```

In this rendering code, mushroom rendering is tightly coupled with mushroom data because the Mushroom class is both a renderer and a data container. Changes to rendering logic would likely affect data declarations and vise versa.

Rendering and data are loosely coupled for game objects, because each renderer only needs to be passed an instance of the type it expects. It does not care how the data is contained, just that the object provides fields/properties/methods it expects.



Coupling

Code parts that rely on each other heavily are *tightly coupled*.

Code parts that rely on each other only a little are *loosely coupled.*

The tighter the coupling, the more code needs to be updated for a single change.

Coupling

Rendering and **data** are naturally coupled in a game: code that renders needs to know what to render.

Loose coupling would separate rendering logic and data organization as much as possible.

Many used a renderer class in addition to multiple different data container classes.

Dependency Injection

- A good way to balance coupling in big software systems, like games.
- Defining base classes or interfaces lets systems and data containers share requirements without being coupled by implementation details.

```
abstract class GameObject
{
    4 references
    public virtual Texture2D Texture { get; set; }

    2 references
    public virtual Rectangle SpriteRectangle { get; }

    2 references
    public virtual Vector2 Size { get; }

    3 references
    public virtual Vector2 Center { get; set; }

    2 references
    public virtual float Rotation { get; }
}
```

```
partial class Mushroom : GameObject
{
    4 references
    public override Texture2D Texture { get; set; }

    2 references
    public override Rectangle SpriteRectangle =>
        new Rectangle(0, 0,
            CurrentFrame * Texture.Width / FrameCount,
            Texture.Height);

    2 references
    public override Vector2 Size => new Vector2(64, 64);

    3 references
    public override Vector2 Center { get; set; }

    2 references
    public override float Rotation => 0f;
}
```

```
class Renderer
{
    /// <summary>
    /// Renders an IGameObject using a SpriteBatch.
    /// </summary>
    0 references
    public void render(
        SpriteBatch spriteBatch,
        GameObject obj)
    {
        spriteBatch.Draw(
            obj.Texture,
            obj.Center - obj.Size / 2f,
            obj.SpriteRectangle,
            Color.White,
            obj.Rotation, obj.Center,
            1f,
            SpriteEffects.None,
            0f);
    }
}
```

`Renderer` does not care how `Mushroom` stores or calculates the `GameObject` data, just that it provides the data. This is loosely coupled. Changing `GameObject` causes compiler errors where `Mushroom` no longer implements it or `Renderer` tries to use members no longer included. This simplifies changes to the coupling data.

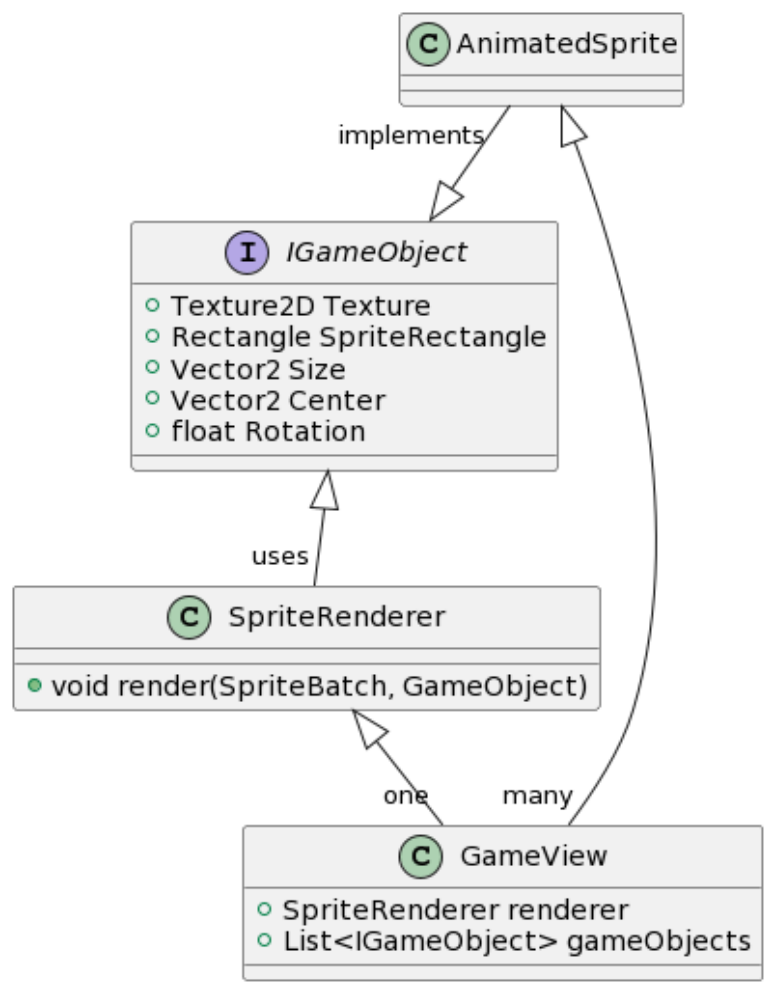
Composition and Inheritance

Many made custom classes that inherit AnimatedSprite, but also used instances of AnimatedSprite to render those custom classes.

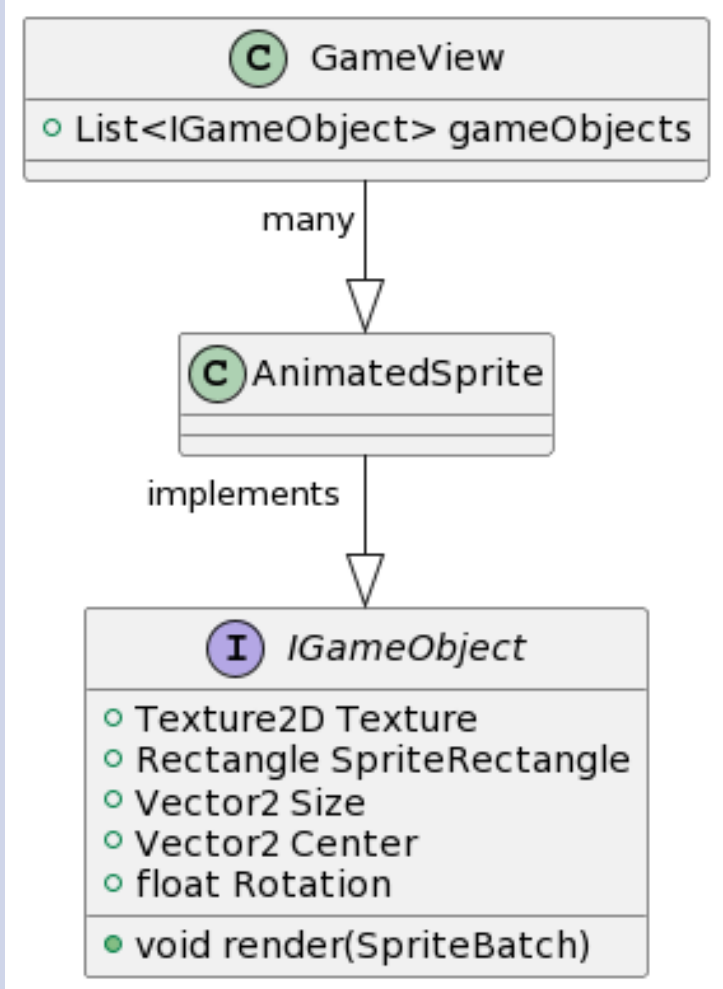
Mixing composition and inheritance without careful design can lead to code that’s deeply coupled and hard to debug or refactor.

Game objects should render themselves (**inheritance**), or be simple data containers collection for a renderer (**composition**).

Composition



Inheritance



Mixed Coupling