

Games as a Mode of Instruction in Object-Oriented Concepts

Krish Pillai

Lycoming College, United States,  <https://orcid.org/0000-0003-0452-4787>

Marcia Lovas

Quinnipiac University, United States,  <https://orcid.org/0009-0003-6749-2257>

Abstract: A typical first computer science course (CS1) introduces the student to coding conventions, variables, methods, control structures, conditionals, and the semantics of classes and objects. Advanced concepts of inheritance, polymorphism, abstract classes, interfaces, and their use in the design process, are covered in a second-level course (CS2). CS2 concepts are abstract, requiring reinforcement through considerable practice. It has been observed that traditional CS2 projects fail to capture the imagination and enthusiasm of students and are seldom useful past the end of the semester, yet interesting projects drawn from the natural sciences may be either too complex or too algorithmic to facilitate the required design experience. Game programming, in contrast, is purpose-driven and has great appeal. Unfortunately, popular game engines hide the engine's complexity and provide too much built-in functionality, relegating the user to writing glue logic in a scripting language. What is needed instead is a challenge that will provide transferable skills for solving generic problems using a statically typed language. The authors of this paper describe a Java game engine and lesson plan they developed for one semester of object-oriented instruction for students who have completed CS2. Early anecdotal results demonstrate that students find the approach challenging, informative, and incentivizing.

Keywords: Graphics, 2D Gaming, Object-oriented Programming, Inheritance, Polymorphism, Event-driven programming, Sprites, Animation, Java.

Citation: Pillai, K. & Lovas, M. (2023). Games as a Mode of Instruction in Object-Oriented Concepts. In M. Shelley, V. Akerson, & M. Unal (Eds.), *Proceedings of IConSES 2023-- International Conference on Social and Education Sciences* (pp. 89-108), Las Vegas, NV, USA. ISTES Organization

Introduction

Object-oriented decomposition is as much an art as it is a science. A decomposition paradigm is an effective strategy for organizing a program in terms of its structure and its functionality. A problem dissected the wrong way can result in an implementation that is needlessly complex and closed-ended. The abstract nature of the design process, and the lack of compiler assistance in matters of design, make it an engineering problem that is hard to learn and arguably complex to teach. The fact that there are multiple structural and behavioral solutions to the same problem makes it imperative that the student is trained on a wide range of projects with clearly

defined requirements specifications. To become an effective software engineer, not only is it important to understand and apply algorithmic decomposition of the problem, but one should also be able to carry out object-oriented decomposition to produce reusable code. Conventional curricula in software engineering have focused on structured analysis. While this is important, exposure to a wider range of challenges involving object-oriented decomposition is valuable as well.

Software Engineering has evolved in a way to encapsulate algorithmic decomposition effectively. As an example, the collections classes provided by Java effectively use overloading to present the most effective algorithm for the task at hand. The `sort()` method provided by the Arrays implementation uses quicksort when the argument is an array with a base type that is primitive but switches to mergesort when the array elements are of a reference type. The onus of choosing the correct algorithm is delegated to the compiler, and it binds the call to one that guarantees a linearithmic response versus one that does not. The algorithmic choice is automated through overloading based on the data type. On the contrary, virtually no support is provided by the compiler or the linker-loader when it comes to object-oriented decomposition. Solutions are mostly problem-specific and optimization issues are left to the programmer.

Application Domains

Software engineering course offerings have conventionally relied on projects drawn from applied mathematics and the natural sciences to provide design and developmental experience to students. In addition to reinforcing theoretical concepts, projects drawn from the natural sciences serve well to prepare students for careers in computational sciences and engineering. However, the implementations for such problems tend to be procedural and algorithmic in nature and seldom present challenges in object-oriented design. For example, the protein folding problem (FAHC, 2023) deals with how a protein's three-dimensional structure is dictated by its amino acid sequence. This engaging problem is essentially a modeling of different small interactions and does not require a hierarchical representation. Another classic problem in astronomy that is interesting to students as a software engineering project is the modeling of Lagrange points ("Lagrange point," 2023) in the vicinity of the gravitational field of two bodies. Again, as before this project presents an issue of algorithmic decomposition rather than object-oriented decomposition through inheritance and polymorphism. Engaging projects such as the ones mentioned above, generally do not lend themselves to challenging issues in the topics that the student ought to be focused on. While object-oriented projects certainly exist in the natural sciences, many require a higher level of preparation in the sciences than what a sophomore student would be expected to know. While conventional problems challenge the student on an accurate implementation of the model of computation, they inadequately expose the student to the concept of decomposition, which is an entirely independent paradigm.

Gaming Domain

The gaming domain has several advantages. Applications today are primarily meant for interactive use and involve use cases that are considerably more complex than what was traditionally the case. Game programming

likewise demands interaction, clearly defined use cases, and complex state changes to manage. In addition, developing a clear knowledge of the game engine framework and its activity lifecycle for the purpose of coding a practical solution should prepare the student to develop software using more advanced frameworks such as those used in mobile platforms. Guiding students in developing their own game engine has suggested positive results such as in Gestwicki's game development efforts (2008). Having students develop their own games utilizing a game engine framework is a natural extension of this effort and may provide even more opportunities for reuse. Additionally, developing games versus developing a game engine provides more creative opportunities and is less restrictive.

Using a game framework to teach software engineering translates the classroom into a game-based learning environment. Game-based learning is a growth area according to a recent study by Metaari (2019), formerly Ambient Insight, a market research firm that uses predictive analytics to gauge trends. Their studies predict that "the worldwide five-year compound annual growth rate (CAGR) for Game-based Learning products and services is a healthy 33.2% and revenues will more than quadruple to reach well over \$24 billion by 2024" (p. 20). It is also worth noting that many programmers cite writing games as their gateway into the field of software engineering. A game programming experience could similarly propel new students deeper into the field.

A game framework when used as a teaching tool has the potential to make learning Java fun and engaging as well as highly effective. Students may retain information longer if the code they write is captivating and useable beyond the semester. By creating games, the student is required to apply their understanding of concepts in a practical way. Games are also shareable and visual, which may be helpful for students of this generation and those who prefer a visual learning style. Sharing generates constructive feedback, helps sustain enthusiasm for the topic, and helps concretize concepts that can be abstract. It can translate classroom learning into a social experience.

Existing Game Engines

Commercial off-the-shelf (COTS) game engines such as Unreal™ (2023) or Unity™ (2023) enable the development of sophisticated games. Unfortunately, such platforms require minimal scientific or mathematical skills, or conceptual expertise from the developer. Designed primarily for artists and the entertainment industry, these engines are heavily automated and reduce developers' contribution to a few lines of glue logic written in C#, JavaScript, or Python. The focus of such engines is on rapid prototyping and development of the game narrative and management of assets. As a result, such off-the-shelf game engines that are highly specialized for game development are unsuitable for imparting software engineering principles. A strong foundation in software engineering, which goes beyond scripting, is essential to be skilled as a software engineer. There is a need for a minimal framework that supports 2D animation and is built on freely available software. A lightweight framework that can run efficiently on inexpensive hardware is what is required for classroom-wide deployment.

Robocode (2023), which was developed by IBM is one such lightweight platform and an excellent framework

for teaching code development. But the use cases it provides are restrictive and might not appeal to students with a variety of interests. In Robocode, the student overrides methods in a framework-provided robot base class to modify the functionality of battle tanks that compete against each other on a battlefield. The final product is not a stand-alone game, but a byte-code module that can operate only within the context of the battlefield. The Robocode framework, though a good demonstration of the use of Java, is limited in its scope as a software engineering teaching tool. It does provide a flexible platform to study Artificial Intelligence and other adaptive behavioral mechanisms, but in terms of inheritance and polymorphism, the scope of the platform is limited. Robocode is ideal for the study of behavioral design patterns when it comes to modeling the behavior of the robot. However, the problem domain is still the battlefield and its appeal to the student group can be limiting.

Java Engine for Teaching Design of Object-oriented Games (JETDOG)

The Jetdog 2D game framework presented in this paper produces stand-alone games and can be used to generate 2D games of a wide variety of genres. The language chosen for game development is Java, the College Board® recommended language of instruction for the Advanced Placement (AP®) Computer Science (2023) course and exam. The game engine was implemented using Java 19 and Swing/AWT, all provided as part of the Java SE distribution. It runs on all platforms, Windows, Mac, and Linux, without the requirement for any native libraries. The framework uses a combination of event detection, sound effects, and gameplay mechanics to reinforce the Java concepts being taught. The game framework is scaled so that games can be developed within a realistic timeframe of one semester. The framework does not replace the skills of the developer but serves to augment them. It is not too complex to learn or master, but at the same time does not require the developer to write extensive code that is not related to the narrative of the game itself. It does not require any expensive hardware such as advanced graphics processing units (GPUs) or specialized libraries such as OpenGL.

A series of lessons are provided that focus on creating content that teaches Java, rather than worrying about the technical details of building a game from scratch. One example of the game framework in action is a lesson where students learn about animations and collisions by creating a game where an alien seeks out a cookie and consumes it. By using principles of trigonometry, students can make the alien move toward the cookie, visually making the concept of directed movements and vectors tangible and easier to understand. Another example is a lesson on object-oriented programming where students create a game where they control a spaceship and shoot down advancing space aliens. By using object-oriented principles like inheritance and polymorphism, students can create a complex and dynamic army of aliens while also learning important programming concepts.

Game Engine Methodology

The game engine handles the rendering of entities on screen, manages their positional updates, and keeps track of various events that occur during a game. A timer clock is the heartbeat of the game engine. Each time the clock ticks, a frame is generated and displayed on the screen. This cycle decides the frame rate of the game

engine. The game engine repositions and renders entities once every frame, and all processing for a specific frame must be completed before the next frame can be processed. As a result of positional changes on the screen, entities may collide with each other, drift out of the visible area of the screen, or run into impenetrable barriers. When they occur, such events are collected and communicated to the game implementation during each cycle of the game loop. Additionally, the keyboard must be polled periodically to check for user input as well. All these actions and event processing are carried out for every clock tick of the game.

The Game Loop

In the simplest sense, all games modeled by this framework consist of a cyclic invocation of a sequence of methods until some conditions are met. The game controller instance moves the game through its various states. A game is essentially composed of a sequence of frames that are generated at a stipulated frame rate (60 frames per second), like a flip book. To enable this strict periodicity, the framework maintains a game clock that fires repeatedly at the stipulated time interval. The clock invokes various methods in the proper sequence to move the game forward. Each frame displays various entities on the screen, to which the user responds through mouse and keyboard actions. The user inputs affect the attributes of each entity on the screen.

The entities may move unobstructed within the viewable screen bounds or may go out of bounds, get blocked by barriers, or the entities may interact with each other through collisions. The way these events are resolved is implemented by the design of each specific game and is not part of the framework. The narrative of the game may even involve the inclusion or removal of entities from the next frame to be computed. As a result of their behavior and interaction with the user and other entities, entities may change their trajectories, exit the game, wrap around, or disappear beyond screen bounds. Before every frame can be computed and displayed, the game must go through all entities known to it, relocate them on the screen based on the elapsed time, and check for collisions, blockages, or out-of-bound situations as shown in Figure 1.

The invocation of various methods is the responsibility of the game clock which fires periodically at 16.667 milliseconds. The game clock invokes methods on the game controller and the entity model. Each event is collected as an array and supplied as an argument to the respective concrete method. For example, all entities are checked for out-of-bounds events during each cycle of the game loop. An out-of-bound event object containing the entity that stepped out of bounds and the edge that was crossed is created for each out-of-bound event. An array of the events is built and supplied as an argument to the `onOutOfBounds(OutOfBounds[])` method, which the implementor of the game overrides for the specific game being built. The game is designed by overriding all the event-related abstract methods that the game engine invokes during every cycle. What makes one 2D game different from another is the way events are handled. The game engine, therefore, serves to abstract out the commonality across all 2D games.

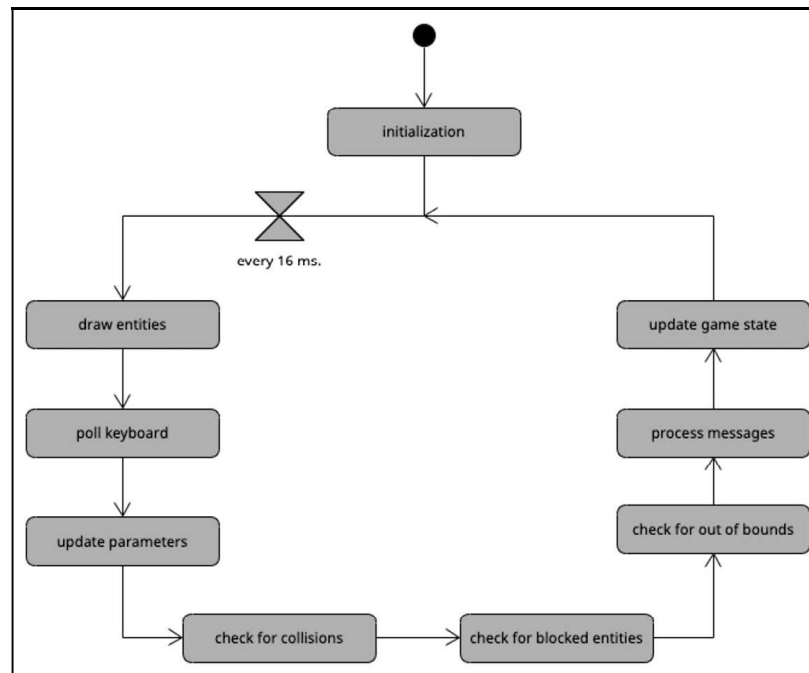


Figure 1. The game engine cycle (or game loop)

Model-View-Controller

The implementation of the framework is based on the Model-View-Controller design pattern. Each entity is represented by a base class called the EntityModel which contains the fundamental attributes that are common across various entities. The controller functionality is provided by the game engine represented by the GameController abstract class. Entities can be of different types, and each of the supported types is represented by the following abstract classes shown in Figure 2:

- Scrolling scenery or background represented by the SceneryModel class
- Animated sprite-based entities represented by the SpriteModel class
- Textual data represented by the TextModel abstract class
- Obstacles or impervious barriers represented by the BarrierModel

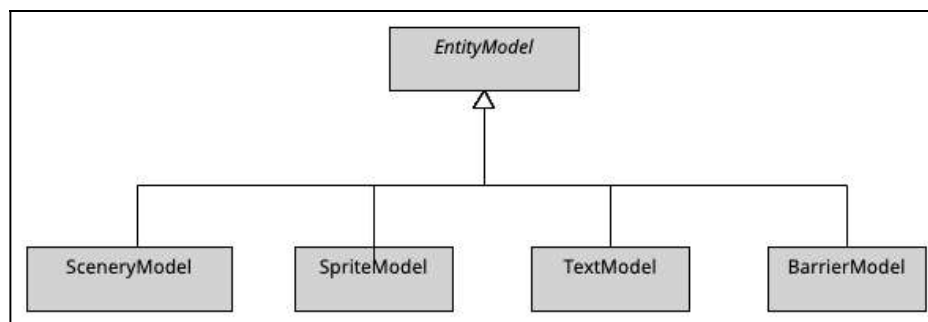


Figure 2. Model hierarchy

Each of these model types contains a reference to its view, which is rendered on screen when the controller invokes the draw() method on the entity. The draw() method is invoked on all entities during each cycle and the call is dynamically bound to the method implemented by the entity instance, which could be a sprite, text, or scenery. The view hierarchy shown in Figure 3 is supported by the base class EntityView, which is further extended into the TextView, SpriteView, and SceneryView classes. Each view type is associated with the corresponding model.

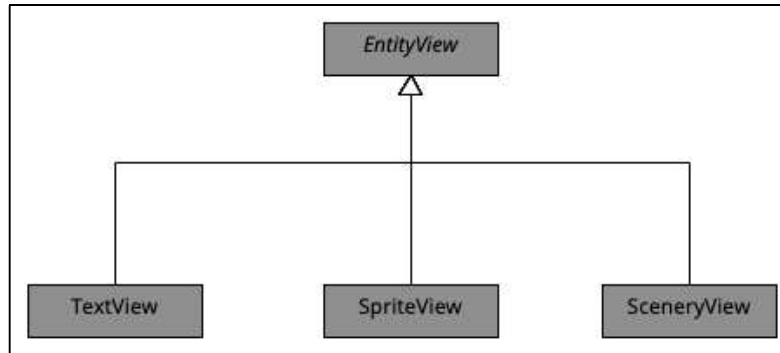


Figure 3. View hierarchy

The game controller, the entity model, and the associated view contribute to building the M-V-C pattern as shown in Figure 4. The controller interacts with the user and updates the model so that the current state of each model is rendered by the view. It serves as a listener for all user interactions. The controller then updates the parameters of each entity based on the elapsed time and user input. This function is carried out for each iteration of the loop.

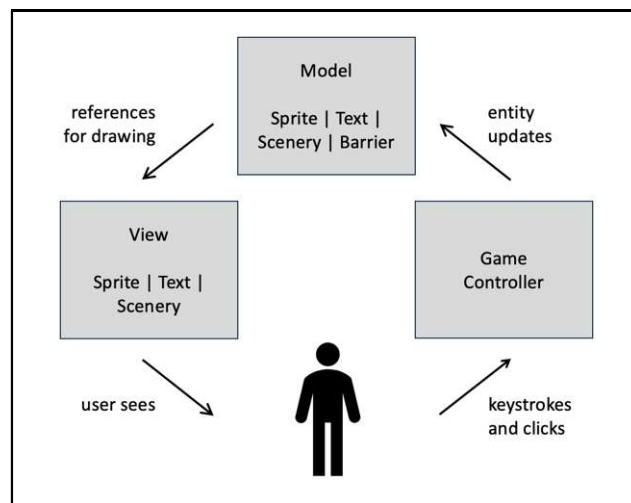


Figure 4. Implementation of the model-view-controller (M-V-C) pattern

A substantial amount of generic functionality is provided by the game controller, freeing up the developer to concentrate on the narrative of the game, which is implemented by providing the concrete implementation for

various abstract methods which are invoked every cycle of the game loop.

Event-Handling

Various events can occur during a game cycle. Entities may go off-screen, they may collide with one another, or be obstructed by a barrier placed on the screen. Entities may even signal to the controller about something that is relevant to the narrative of the game. Classes that represent and manage these events are provided in the event handling framework, which is part of the game engine. Each game handles these events in a specific way that is pertinent to the narrative of the game. The game controller checks all entities for any of these events, aggregates them into arrays of events, and passes them as arguments to the corresponding overridden method. The developer can implement each of these callbacks in the concrete game controller by providing the appropriate handler for each event.

All of the event classes extend the base class `GameEvent` as shown in Figure 5. The events supported by the framework and the handler for each of those events are as follows:

- `CollisionEvent`
 - representing two colliding entities passed as an argument to the `onCollisionEvent()` method
- `OutOfBoundEvent`
 - representing an entity and a screen edge passed as an argument to the `onOutOfBoundsEvent()` method
- `MessageEvent`
 - representing an entity (sender) and a string message (description) passed as an argument to the `onMessageEvent()` method
- `BlockedEvent`
 - representing an entity and a barrier passed as an argument to the `onBlockedEvent()` method

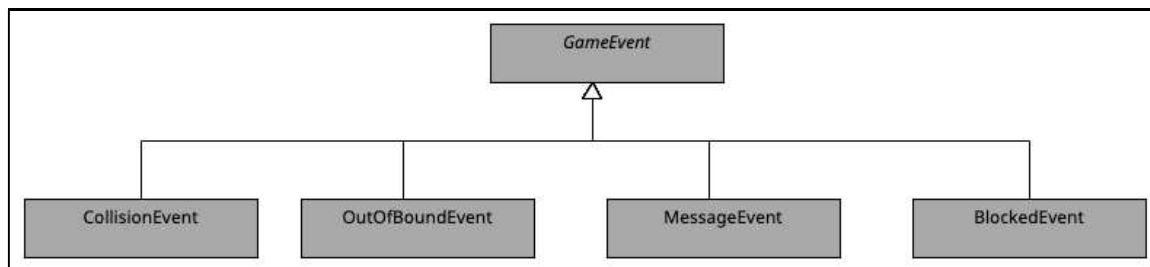


Figure 5. Event classes

Implementing a Game

Writing a game using the framework involves extending the game controller, and the required entity model types, setting the view for those entities, and writing appropriate event handlers for collisions, out-of-bounds,

blockages, and messages. Sounds can be generated by invoking the play() method on a predefined set of enumerations provided by the SoundEffects enum. Textual information, such as a scoreboard can be created by providing a concrete class extension for the TextModel. When a concrete game controller is instantiated, the game clock is started up and the game starts to run. The game clock in turn sets off a sequence of message calls for every cycle. These triggers cause the rendering on screen and various event handlers to be invoked on the game controller instance. The relationship between the various classes is shown in the consolidated class diagram in Figure 6.

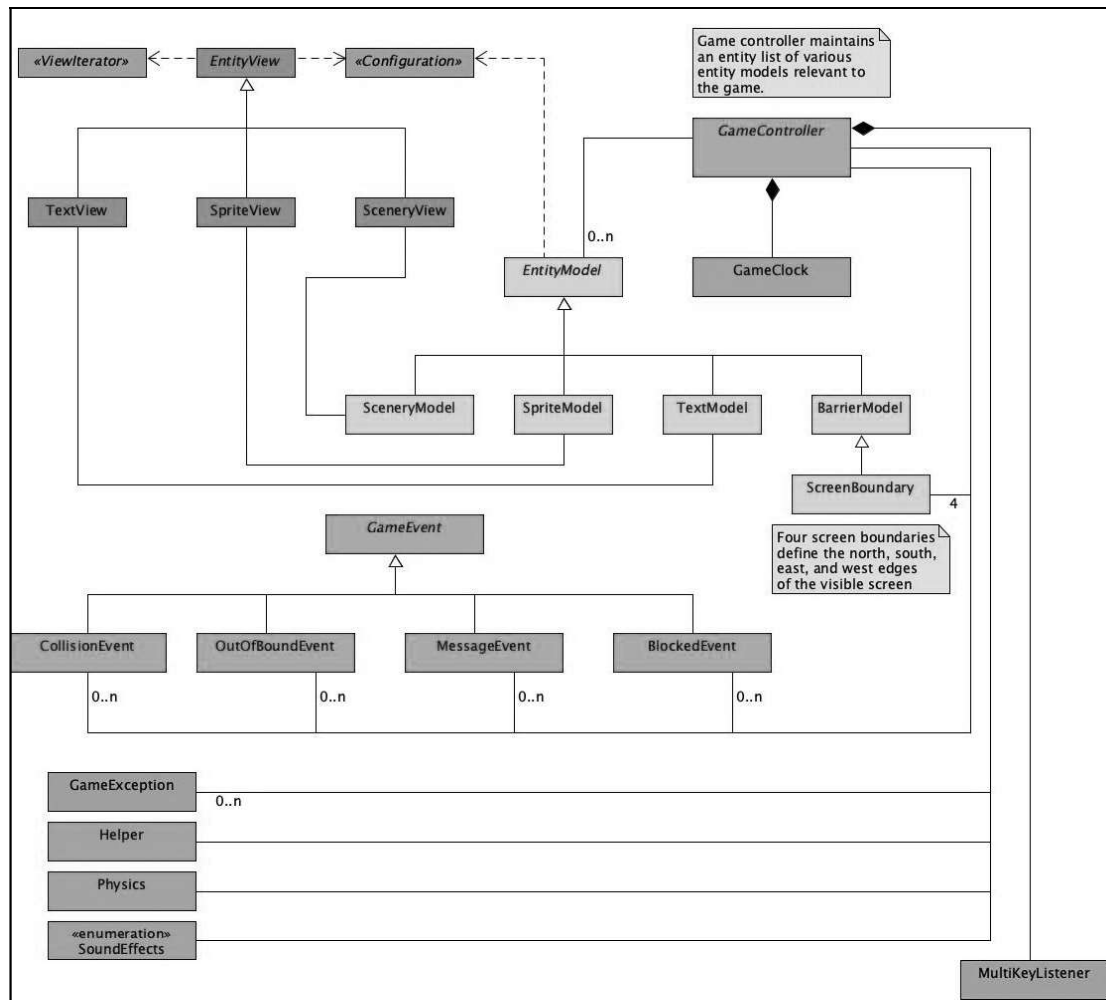


Figure 6. Class relationship diagram

Implementing a game using this framework involves writing derived classes that extend the GameController and the required subclasses of the EntityModel, which could be text, sprite, barrier, or scenery. The GameController provided by the framework is an abstract extension of the Swing JFrame. This gives unlimited access to the developer to all the functionality provided by the JFrame class. Keyboard and Mouse listeners, or any other features supported by the Swing toolkit, are available to the concrete game controller.

Example

Consider a simple 2D simulation of a pair of entities that appear on the screen. Let this class be called TwoBodySimulator as shown in Figure 7.

```
public class TwoBodySimulator extends GameController{

    @Override
    protected void enlistEntities() {
        // place red dot at (100,100) and blue dot at (200,200)
        SpriteModel a = createSpriteModel(Color.RED, 100, 100);
        SpriteModel b = createSpriteModel(Color.BLUE, 200, 200);
        // send them off in different directions
        a.setXVelocity(7);
        a.setYVelocity(8);
        b.setXVelocity(-5);
        b.setYVelocity(-7);
        a.setActive(true);
        b.setActive(true);
        // present them on the screen
        addEntity(a, b);
    }

    private SpriteModel createSpriteModel(Color color, int x, int y) {
        return new SpriteModel(x, y) {
            @Override
            protected void updateParameters(long elapsedTime) {
                // TODO Auto-generated method stub
            }

            @Override
            protected void setAppearance() {
                setView(color);
            }
        };
    }

    @Override
    protected void onCollisionsPolled(CollisionEvent[] events) {
        for (CollisionEvent event : events) {
            EntityModel a = event.getA();
            EntityModel b = event.getB();
            Physics.rebound(a, b);
        }
    }

    @Override
    protected void onOutOfBounds(OutOfBoundsEvent[] events) {
        for (OutOfBoundsEvent event : events) {
            EntityModel em = event.getEntity();
            ScreenBoundary sb = event.getBoundary();
            Physics.rebound(em, sb);
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(()->new TwoBodySimulator());
    }
}
```

Figure 7. Two-body simulator implementation

The objective is to simulate an inelastic collision between the two bodies when they run into each other. Let us also assume that these entities shall bounce off the edge of the screen. The first step is to extend the GameController class. The enlistEntities() method is overridden to add the two entities. Since collisions and out-of-bounds are the two events that need to be defined for this simulator, the onCollisionsPolled() and the

onOutOfBounds() methods are implemented. All other events are stubbed out, and not shown in the listing below for the sake of brevity. A simple class with three or four methods creates the basic simulation.

More elaborate games are feasible by extending classes from the framework and implementing abstract methods on the controller and models. The periodic game clock invokes a series of abstract methods, the concrete versions of which are to be supplied by the user-implemented classes. The rules of the game can be coordinated by the updateGameState() method, which is invoked by the game clock at the end of each clock tick. The sequence in which the game clock invokes various methods on the game controller each time it is fired is shown in the detailed life-line diagram beginning with Figure 8.

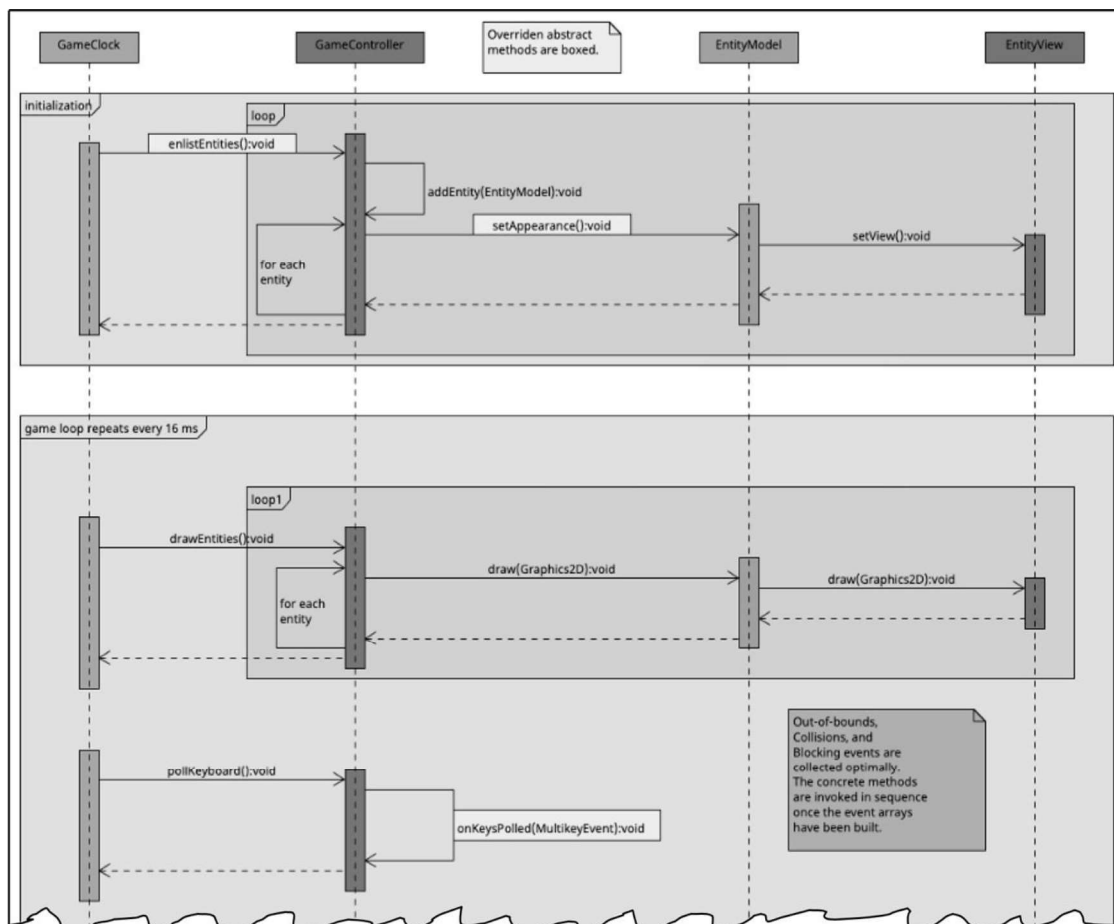


Figure 8. The game-loop sequence: initialization, drawing, and keyboard polling.

The first cycle executed by the timer activity is unique. It starts off by invoking enlistEntities(), where entities that are to be seen on the screen can be added. When each entity is added, the corresponding setView() method is invoked by the controller. This initialization does not happen subsequently. Once the game loop starts off, the timer activity repeats the sequence of calls.

The game loop executes 60 times every second. It starts off by drawing all entities participating in the game on the screen. While the user is visually processing the screen, the game engine proceeds to poll the keyboard for multiple key presses. The polled keyboard functionality is used specifically for in-game interaction. Being a software polled mechanism, a single keystroke that may last more than the cycle time of the game loop will be detected multiple times. If a single keystroke is to be detected, such as in the case of user-initiated game state changes, the recommended approach is to implement the standard `KeyListener` provided by Swing.

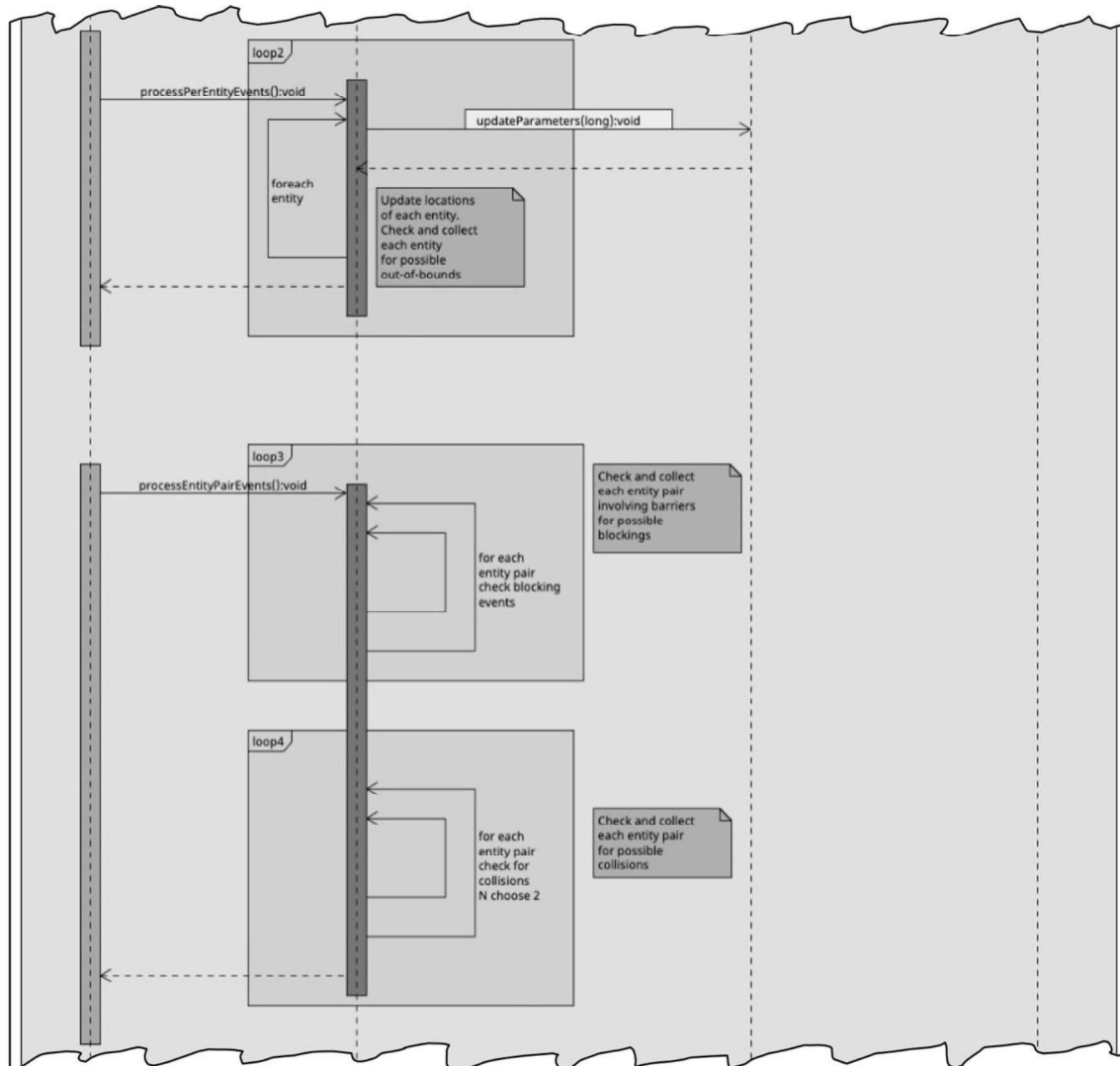


Figure 9. The game-loop sequence: parameter updates, pre-checks on collisions, blocks, and out-of-bounds events.

The next stage of the game loop involves updating the locations of all the entities participating in the frame. This is done by applying the current acceleration and velocity settings to each entity, given their initial locations on the screen. Repositioning the entities on the screen opens the possibility of entities colliding with each other, being blocked by barriers, or straying off the viewport (visual area on the screen). The next steps executed by

the game loop involve detecting these events, collecting the necessary details of each event, and invoking the appropriate overridden methods as shown in Figure 10.

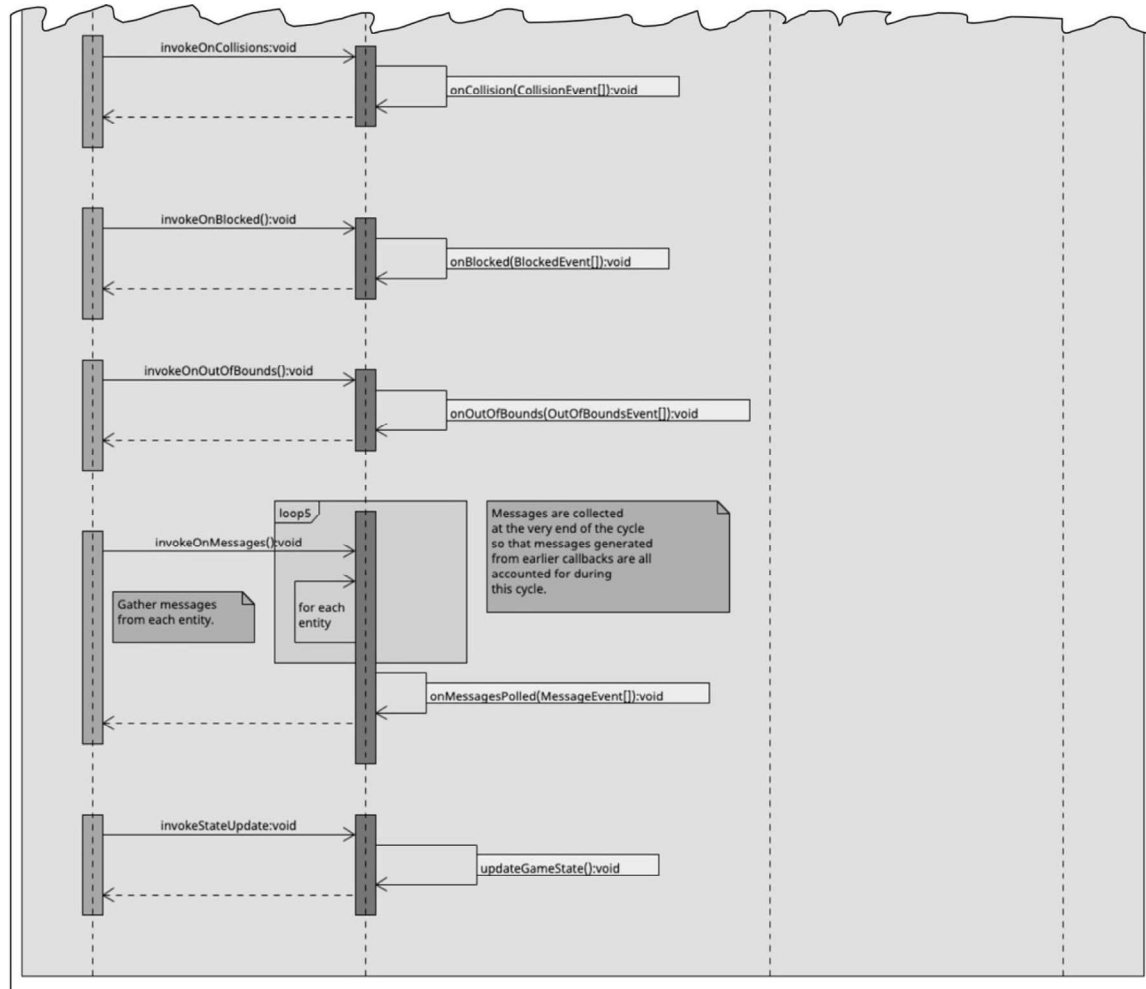


Figure 10. The game-loop sequence: the invocation of overridden methods.

Lesson Methodology

The accompanying lesson plan consisting of seven lessons as shown in Table 1 was designed to introduce the student progressively to the game engine functionality through increasingly complex exercises while requiring them to apply object-oriented concepts as they develop a full-fledged video game.

Table 1. Lesson Plan

Lesson	Game topic(s)	Object-oriented topic(s)
1 Game loop	Game engine design Model View Controller (MVC)	Inheritance, abstract classes, MVC pattern

	Game loop activities	
2 Vector dynamics	Using trigonometry and vector operations to navigate on screen	Event-handling, using an interface
3 Gravity and collisions	Using acceleration and inelastic collisions	Inheritance, abstract classes
4 Guided projectiles	Event-driven creation of entities that follow a defined trajectory	Anonymous inner types
5 Composite movements	More complex interactions	Aggregation
6 State machines	Using a state machine to model a multi-level game	State pattern
7 Executable app	Full Space Aliens game	Polymorphism

The lessons start with simple animations, then delve into simple vector dynamics, game physics, more complex movement and behavior, and state machines. The final lesson is a game that is fully functional and can be run as a stand-alone application.

Results

In an experimental offering of this course using a prototype game engine and lesson plan in the spring of 2022 at Lycoming College, the students built a Space Aliens game, shown in Figure 11, similar to Space Invaders™, that could run on any desktop.

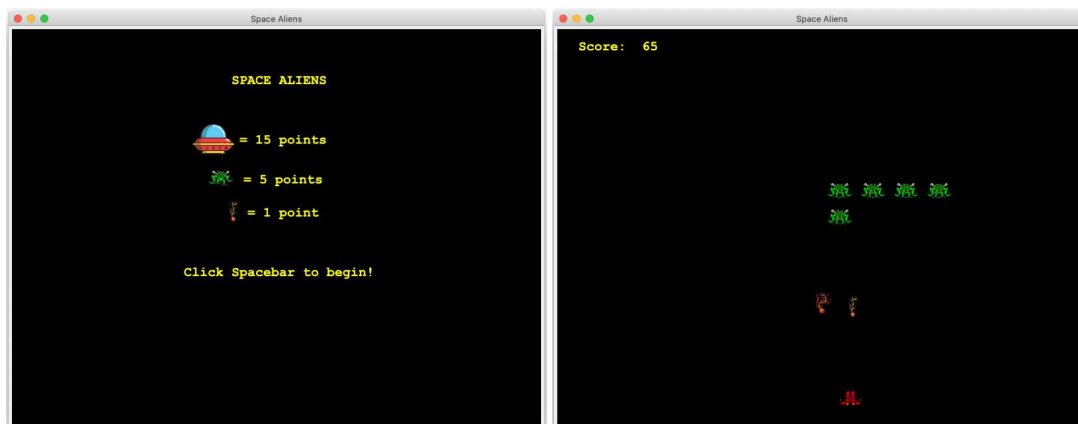


Figure 11. Student game implementation

Upon completion of the experimental course, the class of six students was provided with a questionnaire, and the results are shown in Figures 12 and 13.

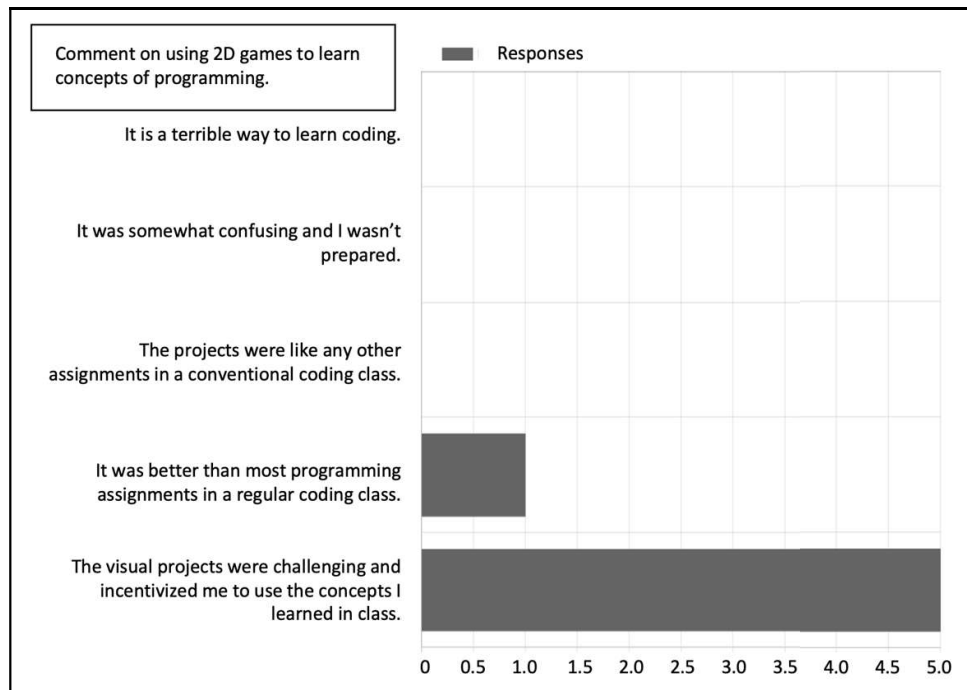


Figure 12. Student responses to 2D Game Design as a medium of learning

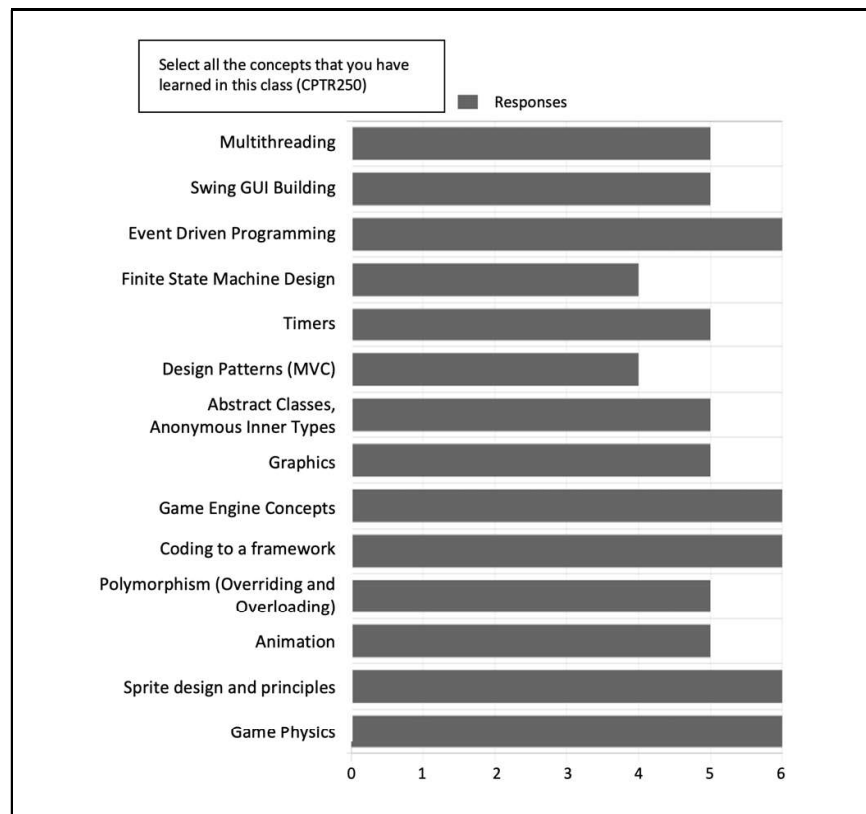


Figure 13: Student responses to concepts learned

Discussion

A majority of participating students found this game-based learning experience challenging and incentivizing. It was also apparent that the game engine-based programming class covered a wide range of topics in software engineering. A majority of participating students found the course informative on event-driven programming, coding to a framework, and the object-oriented concepts of inheritance, polymorphism, interfaces, abstract classes, and anonymous inner types.

These reviews were positive, but the sample size was not large enough for experiments beyond the small observational study. In addition, it is worth noting that the six students in the course had all taken a second-level course in computer programming (CS2). Four of the six participating students were intermediate to advanced-level programmers and as such, the feedback received from the group may not correlate to how a lesser-prepared class might fare.

Conclusion

Game programming with the JetDog engine demonstrates an accessible way to teach object-oriented design topics, perhaps more easily so than other practical problem domains. The game framework was anecdotally shown to be a motivating and effective way of teaching Java. Students were observed to be more engaged and enthusiastic than in traditional Java courses, and students found the course challenging and informative on a variety of software topics. JetDog provides a richer environment than off-the-shelf gaming solutions for practicing coding to a framework, an experience sure to benefit students as they enter the field of software engineering.

Recommendations

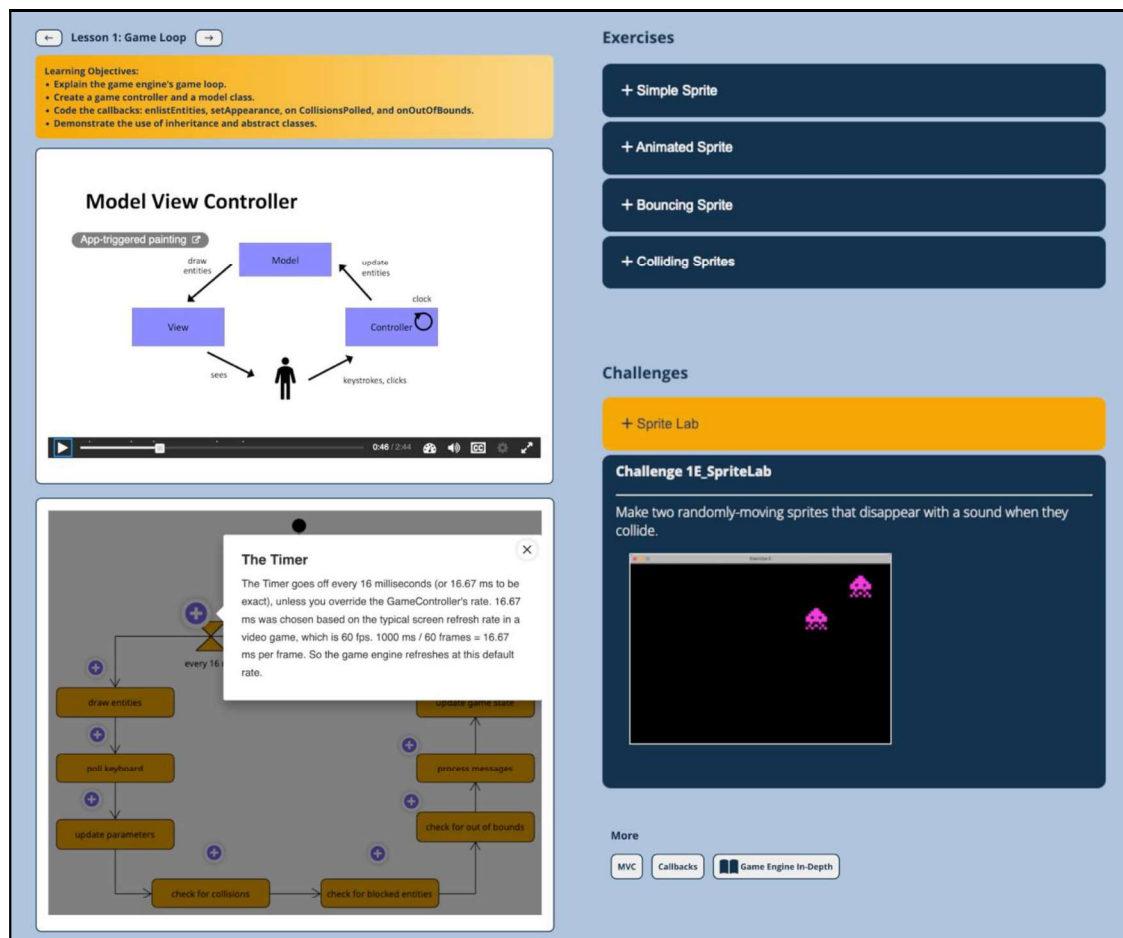
Going forward, there is room for augmentation and refinement of the game engine and lesson plan, as well as collaboration with instructors and their students to further evaluate the effectiveness of the JetDog framework as a teaching tool.

The lessons may be expanded to cover more than one final game project. It is observed that students become more creative when the game they are building is something with which they identify. Additional exercises for games similar to Pong™, Asteroids™, and other games would cater to the varying tastes of students as well as provide projects for different semester offerings of the course. Adding grid placement functionality to the game engine, as described in Purewal's framework (2006), would provide students the capability to practice mapped spaces and polymorphism through the implementation of games similar to Minesweeper™, Snake™, or Tetris™. The game engine and lessons may also be improved based on feedback from students and collaborating instructors.

A collaborative study assessing students' ability to apply object-oriented concepts in Java before and after this course would be the next step in the research. Collaboration is sought with instructors having access to larger classrooms (25+ students), who, in exchange for using the game engine and lessons, would administer the necessary assessment instruments required to collect data for further study on the approach's effectiveness. Collaborating instructors' students would have access to a website containing the lessons, instructional videos, schematics, and references for in-depth exploration of the topics. In addition, the game engine framework is supported by detailed hypertext documentation using Javadoc.

Delivery

Delivery of the lessons is an important part of engaging students in an active way. A visual, engaging website will be used for lesson delivery. Each lesson introduces the student to a specific topic through a video, interactive HTML, or simulation content and offers self-test exercises and laboratory challenges to develop their understanding of the concepts. The general format of a lesson is shown in Figure 14.



Lesson 1: Game Loop

Learning Objectives:

- Explain the game engine's game loop.
- Create a game controller and a model class.
- Code the callbacks: `enlistEntities`, `setAppearance`, `onCollisionsPooled`, and `onOutOfBounds`.
- Demonstrate the use of inheritance and abstract classes.

Model View Controller

App-triggered painting (off)

draw entities (Model to View)

update entities (Controller to Model)

clock (Controller to Model)

sees (View to Player)

keystrokes, clicks (Player to Controller)

The Timer

The Timer goes off every 16 milliseconds (or 16.67 ms to be exact), unless you override the GameController's rate. 16.67 ms was chosen based on the typical screen refresh rate in a video game, which is 60 fps. $1000 \text{ ms} / 60 \text{ frames} = 16.67 \text{ ms per frame}$. So the game engine refreshes at this default rate.

Game Loop Flowchart:

```

graph TD
    Start(( )) --> Draw[draw entities]
    Draw --> Poll[poll keyboard]
    Poll --> Update[update parameters]
    Update --> Collisions[check for collisions]
    Collisions --> Blocked[check for blocked entities]
    Blocked --> OutOfBounds[check for out of bounds]
    OutOfBounds --> Process[process messages]
    Process --> GameLoop[restart game loop]
    GameLoop --> Draw
  
```

Exercises

- + Simple Sprite
- + Animated Sprite
- + Bouncing Sprite
- + Colliding Sprites

Challenges

- + Sprite Lab

Challenge 1E_SpriteLab

Make two randomly-moving sprites that disappear with a sound when they collide.

More

- MVC
- Callbacks
- Game Engine In-Depth

Figure 14. Lesson structure: interactive elements, exercises, challenges, and resource links

Lesson 1, as an example, covers sprite creation and animation. The first exercise in this lesson involves putting a simple sprite on the screen. The next exercises introduce the student to the use of callback methods to make the sprite entity move and bounce off the edges of the screen. Once the student has learned to control the behavior of the entity, the lesson provides more advanced exercises and a challenge, requiring the modeling of two colliding sprites with sound effects.

Early exercises in each lesson are scaffolded with coding hints as shown in Figure 15. The student is provided with a cross-reference to the sequence diagram as shown in Figure 16 so that their code is introduced at the proper state of the game engine. In addition, links to pertinent areas of the Javadoc API are also supplied with each exercise as shown in Figure 17.

```

• Here's a start:

@Override
protected void onCollisionsPolled(CollisionEvent[] events) {
    for (CollisionEvent ev : events) {
        // add your code here
    }
}

@Override
protected void onOutOfBounds(OutOfBoundsEvent[] events) {
    for (OutOfBoundsEvent event : events) {
        switch (event.getEdge()) {
            case North:
            case South:
                // add your code here
                break;
            case East:
            case West:
                // add your code here
                break;
            default:
                break;
        }
    }
}

```

Figure 15. Coding hints

The website may be used in a traditional or flipped classroom mode, in which students have the opportunity to become familiar with the material outside of class and control their learning. In flipped mode, class time can be utilized for live problem-solving, targeted discussion, and interaction based on students' pre-work.

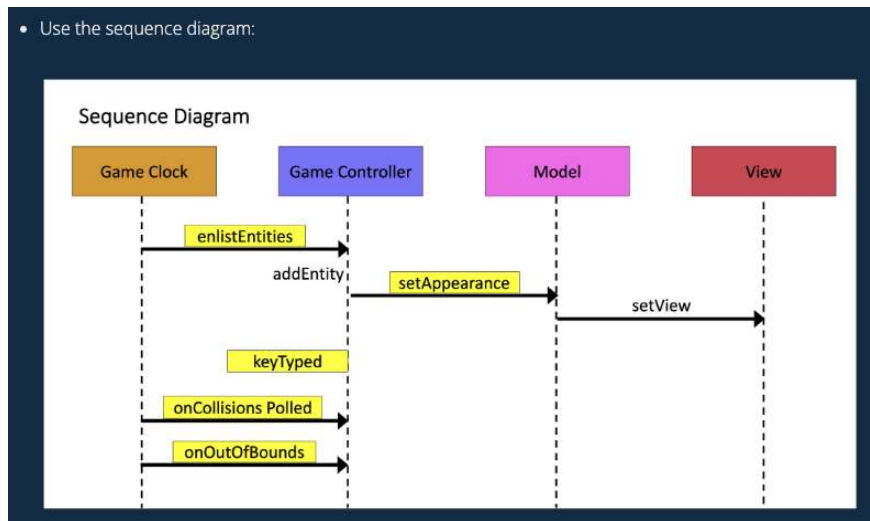


Figure 16. Reference to the applicable sequence diagram

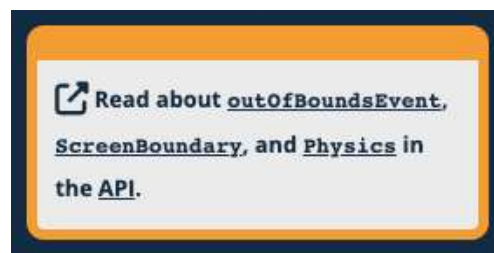


Figure 17. API links

Acknowledgments

Thank you to the students in the spring 2022 CPTR 250 course for their excellent work on the lessons, and their positive feedback. Thanks to the Mathematics Sciences Department for the opportunity to execute a trial run of this course. Thank you to Mavin James for their hard work and dedication in completing their final project and providing valuable feedback.

References

- AP Computer Science A Course – AP Central | College Board.* (2023). College Board Website. Retrieved July 23, 2023, from <https://apcentral.collegeboard.org/courses/ap-computer-science-a>
- Gestwicki, Paul, & Sun, Fu-Shing. (2008). Teaching Design Patterns Through Computer Game Development. *ACM Journal on Educational Resources in Computing*, (Vol. 8, No. 1, Article 2).
- Lagrange point. Celestial mechanics and the three-body problem (2023, July 20). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Lagrange_point&oldid=1166264282
- Metaari. (2019). The 2019-2024 Global Game-based Learning Market: Serious Games Industry in Boom Phase.

Serious Play Conference (August 1, 2019).

Purewal, T. S., & Bennett, C. (2006). A framework for teaching polymorphism using game programming. *Consortium for Computing Sciences in Colleges: Southeastern Conference*, (JCSC 2022, 2 December 2006).

Robocode. (2023, February 26). Robocode Homepage. Retrieved July 23, 2023, from <http://robocode.sourceforge.io>

The Folding@home Consortium (FAHC). (2023, July 25). *Start Folding – Folding@home*. Retrieved July 25, 2023, from <https://foldingathome.org/>

The most powerful real-time 3D creation tool- Unreal Engine. (2023). Unreal Engine Homepage. Retrieved July 23, 2023, from <https://www.unrealengine.com/>

Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. (2023). Unity Homepage. Retrieved July 23, 2023, from <https://unity.com/>