Exercise 4

Software Development 2019
Department of Computer Science
University of Copenhagen

Niels-Christian Borbjerg <nibo@di.ku.dk>,
Jens Kanstrup Larsen <jkl@di.ku.dk>

Version 1: February 22th

Due: March 1st, **15:00**

Abstract

This exercise set 4 is a **group exercise. There are no individual elements in this exercise.**

The deliverable will consist of a report, and a code hand-in. The report *must* be submitted through Absalon, and the code *must* be submitted through `github`.

1 Installing DIKUArcade

The game engine `DIKUArcade` is located at:

https://github.com/diku-dk/DIKUArcade

1.1 Create a `git`-Repository

In order to get started using `DIKUArcade` you first need to create a private repository to contain the solutions to the exercise set(s). This should be done as the following

Create private repository

Navigate to your `github` home page. Create a new empty, private repository with the name `"su19-<your-short-group-name>"`. Do not add a README-file or a .gitignore.

Create a directory on you own machine

Open a terminal and navigate to a suitable location on you computer. Then enter the following commands to connect with your newly created repository:[1]

[1] Windows-users cannot use this directly in the command-prompt, but may instead use git bash.

```
$ mkdir su19-<your-short-group-name>
$ cd su19-<your-short-group-name>
$ git init
$ echo "# SU19 Repository" >> README.md
$ git add README.md
$ git status
$ git commit -m "Initial commit"
$ git remote add origin <url-for-your-private-repository>
$ git push -u origin master
```

Add `DIKUArcade` as a `git` submodule

To get started on your own Rider projects, you will need `DIKUArcade` contained in your repository. The `git` feature "submodule" will allow you to import `DIKUArcade` to your repository without it being a part of your `git` history. Same as before, some terminal commands will help you with this setup:

```
$ git submodule add https://github.com/diku-dk/DIKUArcade
$ git submodule init
$ git submodule update --recursive --remote
```

The last of these commands should be used regularly, as it will update your local copy of `DIKUArcade` . In the case of an update to our game engine, you will be sure to always have the latest version installed.

Adding your TA

Now, navigate to your repository home page and add your TA in the Settings > Collaborators section. Unless you do this, you will not get feedback for this exercise, nor will you get points. Ask your TA during exercise sessions for his/her github account name.

1.2 Create Rider Solution

Now that you have a `git`-repository set up for this and future exercise sets, it is time to get started with the programming part. Open Rider from the repository folder and create a new solution with the name `"SU19-Exercises"`.

1.2.1 Adding `DIKUArcade` to your solution

Right-click the Solution in the 'Solution Explorer' and select `Add Existing Project`. Find the `DIKUArcade.csproj` file from the `git` submodule path and select this.

Restoring `DIKUArcade` packages

Right-click the `DIKUArcade` project folder in the 'Solution Explorer' and select `Manage NuGet Packages`. From this menu find the list of installed packages - select `OpenTK * 2.0.0` and click on the uninstall button. After this is done enter `OpenTK` in the search field, click on the exact same package and click to install it again. This step is, unfortunately, necessary due to the way that Rider works together with `git`. However, it is a small fix, and will enable you to compile the game engine.

1.3 Create a Rider Project

Now that you have a solution with `DIKUArcade` attached to it, you will need to create a Rider Project for this exercise set.

Create a new project with the name `"Galaga-Exercise-1"`. Make sure to reference `DIKUArcade` so that you can use the engine.

Read the rest of the Assignment Text before beginning to code

Now, make sure to read the entire assignment text before you begin coding - this exercise is a bit bigger than the ones you have solved in the previous weeks, so make sure that you don't leave out any part in your solution.

2 Game Engine

`DIKUArcade` is a 2D game engine, which means that is has support for the most common features needed for at 2D game. This includes reading images from files, creating an application window, event processing, game entities, rendering, timers, etc.

2.1 Coordinate System

Rather than specifying object positions on-screen using pixel-coordinates, we use a normalized coordinate system with x, y axes in the range $[0, 1]$ as shown in figure 1. This has the benefit of objects on screen keeping the same relative size and distance to each other across different screen resolutions.

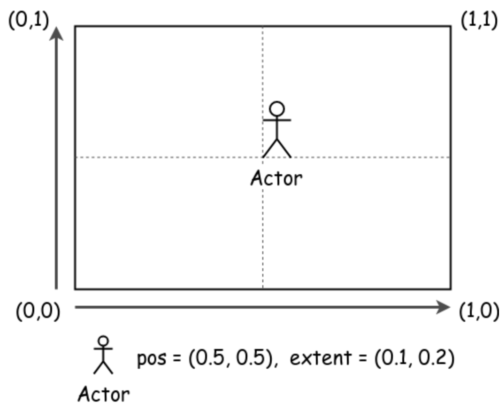

Figure 1: DIKUArcade uses a normalized coordinate system.

2.2 Engine Structure

`DIKUArcade` is structured into several namespaces to separate concerns and increase decoupling. Here, we list the most important, which will be used in this exercise:

`Entities :` classes for structuring game objects, and placing them into generalized containers which can be iterated.

`EventBus :` a set of data structures to support processing of various game events.

`Graphics :` classes concerned with drawing objects onto the screen. This namespace depends heavily upon the `Entities` module.

`Math :` for now this namespace only contains vector types, which are used in other parts of the engine.

`Physics :` a predefined algorithm for performing collision detection. Will later be extended to gravitational forces, mass, and other physics-related game programming needs.

`Timers :` currently only 2 classes - a stopwatch and a game speed timer.

3 Galaga

In this exercise set, you will create a simple, but functioning version of the classic arcade game `Galaga` [2]. In figure 2 you can see how the game is supposed to look like after this exercise has been completed.

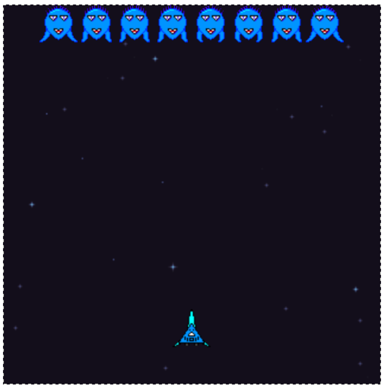

Figure 2: Exercise 4 milestone

3.1 Getting started with a Game class

To get started with creating the game, we will consider this setup:

3.1. Create a `Program.cs` file *in your new project for this exercise*. This is where your entry point should be placed.

3.2. Create a `Game.cs` file *in the same location*. This file should contain only a single class, and the recommended setup is shown in the code snippet below:

(Tip: In Rider, press Alt + Enter *to navigate to the definition of the method/field you have selected.)*

```
using DIKUArcade;

public class Game : IGameEventProcessor<object> {
    private Window win;
    private DIKUArcade.Timers.GameTimer gameTimer;

    public Game() {
        // TODO: Choose some reasonable values for the window and timer constructor.
        // For the window, we recommend a 500x500 resolution (a 1:1 aspect ratio).
        win = new Window(... , ... , ...);
        gameTimer = new GameTimer(... , ...);
    }

    public void GameLoop() {
```

[2]https://en.wikipedia.org/wiki/Galaga

```
        while(win.IsRunning()) {
            gameTimer.MeasureTime();
            while (gameTimer.ShouldUpdate()) {
                win.PollEvents();
                // Update game logic here
            }

            if (gameTimer.ShouldRender()) {
                win.Clear();
                // Render gameplay entities here
                win.SwapBuffers();
            }

            if (gameTimer.ShouldReset()) {
                // 1 second has passed - display last captured ups and fps
                win.Title = "Galaga | UPS: " + gameTimer.CapturedUpdates +
                            ", FPS: " + gameTimer.CapturedFrames;
            }
        }
    }

    public void KeyPress(string key) {
        throw new NotImplementedException();
    }

    public void KeyRelease(string key) {
        throw new NotImplementedException();
    }

    public void ProcessEvent(GameEventType eventType,
        GameEvent<object> gameEvent) {
        throw new NotImplementedException();
    }
}
```

The size of the code may seem overwhelming at first, but you should understand the following:

- The `GameLoop` method handles all game updates and rendering.
- The `Window` class is responsible for drawing the actual window on screen. Its constructor takes a window name and size/resolution as arguments.
- The `GameTimer` is a class that can consistently signal when the game logic should be updated (`ShouldRender` returns `true`), and when the game's rendering should be refreshed (`ShouldUpdate` returns `true`).

  You may wonder why we are not simply using a while-loop or variant thereof to control update speed. The reason for this is that different machines with different processing power may run a different number of while-loop iterations each unit of time, and we want to make sure that the number of updates each unit of time is consistent across systems. Practically speaking, we wish to put a cap on the number of frames per seconds (FPS) and the number of updates per second (UPS). The `GameTimer` lets us do this by letting us check `ShouldRender` and `ShouldUpdate` respectively.

The number of updates per second must be 60, but you are free to use any amount of FPS that you see fit. (Remember that most screens have a capped refresh rate of 60 Hz, and that 10 FPS probably looks really stuttered.)

For now, ignore the `ProcessEvent` and `Key` methods.
Try to input your choice of constructor values, then run the program by creating a `Game` object and calling `GameLoop`. If everything is correct, a black screen should show up. The window has default key bindings: Escape for closing the window, and F12 for taking a screen shot. Verify that everything works as intended up to this point before continuing.

3.2 Creating a Player Entity

In `Galaga` we need some assets, which in this case means images. Therefore, download `Galaga-Assets.zip` from the files on Absalon, and unpack it in the `Galaga-Exercise-1` folder. This folder contains all the images needed for this game.

We are now going to create the `Player` class, which will represent the player's spaceship. To do so, begin by creating a `Player.cs` file with the following class implementation:

```
public class Player : Entity {
    private Game game;

    public Player(Game game, DynamicShape shape, IBaseImage image)
        : base(shape, image) {
        this.game = game;
    }
}
```

Then, follow these steps in the `Game` class:

3.1. Add a private `Player` field called `player`.
3.2. Add the following code snippet to the `Game` constructor:

```
player = new Player(this,
    new DynamicShape(new Vec2F(0.45f, 0.1f), new Vec2F(0.1f, 0.1f)),
    new Image(Path.Combine("Assets", "Images", "Player.png")));
```

We use the `DynamicShape` here, instead of a `StationaryShape`, because it contains a direction vector. This will be needed later.

Note that it is possible to provide an `ImageStride` object for the `Entity` instead of an `Image`, if the entity should be animated. (e.g. a fire texture changing image every 100th millisecond).

3.3. Add a call to `player.RenderEntity()` to the `GameLoop`.

Please verify that the entity is actually being drawn to the screen before continuing.

3.3 Making the player move

We should now make the player able to move. This will be the first step towards a real, playable game experience. In this implementation of `Galaga` the player shall only be able to move *left* and *right*.

First off, we want to be able to update the direction of the `DynamicShape` that is associated with the `player` object. Since the `Player` class should be responsible for handling its own movement, we want to be able to call two public methods that control its movement; one that sets the direction of the player, and one that moves it based on its current direction.

Add a `Direction` method to the `Player` class with the following behavior:

Input: A `Vec2F` vector describing the direction.

Output: `void`.

Behavior: Updates the `Direction` of the internal `Shape` property. *Hint: You may need to look into using the* `AsDynamicShape` *method.*

Add a `Move` method to the `Player` class with the following behavior:

Input: None.

Output: `void`.

Behavior: Calls the `Move` method of the internal `Shape` property, *unless* the player is attempting to move outside the window borders (remember, normalized coordinate system!).

Add the call to `player.Move` in the `GameLoop` where the game logic is updated.

While the player can technically move now, there is no event that properly updates its direction. We want to update this direction depending on keyboard input. To make the `Game` listen for key inputs you will use the `DIKUArcade.EventBus.GameEventBus<object>` class. This class can send notifications on events to certain objects that "*subscribe*" to it. Such objects are the ones that implement the `IGameEventProcessor` interface, like our handed-out `Game` class does. Whenever they receive a notification, they call their `ProcessEvent` method along with arguments describing what event happened. We will use this to implement player movement whenver the correct key is pressed.

Implement movement using the following steps:

3.1. Add a private `GameEventBus<object>` field called `eventBus`.

3.2. Add the following code snippet to the `Game` constructor:

```
eventBus = new GameEventBus<object>();
    eventBus.InitializeEventBus(new List<GameEventType>() {
```

```
            GameEventType.InputEvent,  // key press / key release
            GameEventType.WindowEvent, // messages to the window
        });
win.RegisterEventBus(eventBus);
eventBus.Subscribe(GameEventType.InputEvent, this);
eventBus.Subscribe(GameEventType.WindowEvent, this);
```

3.3. Add a call to `eventBus.ProcessEvents()` to the `GameLoop`. This will make the `eventBus` notify all subscribers.

3.4. Remove the exception throw from `ProcessEvent` and add the following code instead:

```
public void ProcessEvent(GameEventType eventType,
    GameEvent<object> gameEvent) {
    if (eventType == GameEventType.WindowEvent) {
        switch (gameEvent.Message) {
            case "CLOSE_WINDOW":
                win.CloseWindow();
                break;
            default:
                break;
        }
    } else if (eventType == GameEventType.InputEvent) {
        switch (gameEvent.Parameter1) {
            case "KEY_PRESS":
                KeyPress(gameEvent.Message);
                break;
            case "KEY_RELEASE":
                KeyRelease(gameEvent.Message);
                break;
        }
    }
}
```

3.5. Finally, remove the exception throws from `KeyPress` and `KeyRelease` and add the following code to `KeyPress`:

```
private void KeyPress(string key) {
    switch(key) {
        case "KEY_ESCAPE":
            eventBus.RegisterEvent(
                GameEventFactory<object>.CreateGameEventForAllProcessors(
                GameEventType.WindowEvent, this, "CLOSE_WINDOW", "", ""));
            break;

        /*
         * TODO: Add cases that call player.Direction with a suitable direction.
         * You can match on cases such as "KEY_UP", "KEY_1", "KEY_A", etc.
         * Remember that the entire screen is 1.0f wide/tall, so choose a
         * fittingly small number for the direction, e.g. (0.01f,0.0f).
```

```
        */
    }
}
```

Add another direction change when *releasing* the keys[3].

This is quite a mouthful, but you should be safe if you just follow the steps above closely. Please do not continue until you have the player object moving around and staying inside the window.

3.4 Create Enemies

Now that you have a player entity which can move, we will look into how the enemies can be created. These will use an `ImageStride` instead of an `Image`. The construction looks a bit different, due to several reasons:

3.1. `ImageStride` objects need to know how often they should change image.

3.2. Since we a going to create multiple enemies using the same sequence of images, we concern ourselves with the deficiency of reading in the same images once for every enemy object! The fix is to read the images from harddrive *once* and store them in a `List<Image>` container, then reference this list every time we create a new `ImageStride` object.

Follow these steps to insert some simple enemies into the game:

3.1. First, add an `Enemy.cs` file to your project and create a `Enemy` class. Its should inherit from the `Entity` class, and its constructor should be identical to the one in the `Player` class. It is quite bare right now, but the class will be extended for a later assignment.

3.2. Add a `List<Image>` field called `enemyStrides` and a `List<Enemy>` called `enemies`.

3.3. Add the following code snippet to the `Game` constructor:

```
// Look at the file and consider why we place the number '4' here.
enemyStrides = ImageStride.CreateStrides(4,
    Path.Combine("Assets", "Images", "BlueMonster.png"));
enemies = new List<Enemy>();
```

3.4. Add a method called `AddEnemies` which creates the enemies and adds them to the enemy list. You should create the enemies in the same manner as you create the player instance.

The enemies **must** change image stride every 80 milliseconds, and their extent **must** also be `(0.1f, 0.1f)`.

3.5. Update the `GameLoop` so that it renders all objects in `enemies` (use a `foreach` loop).

[3] In the class `DIKUArcade.Input.KeyTransformer` you can see a list of prefedefined key strings. You do not need to instantiate this class or reference to it in any way, as this is automatically handled for you by the game window.

As before, verify that the animations are working correctly before continuing. Remember to call `AddEnemies` in order to actually add the enemies.

3.5 Adding Player Shots

As of now, your Player entity should be able to move around the bottom of the screen, and there should be aliens at the top of the screen "wiggling" away. Now, it is time to implement another key component of Galaga - Player projectiles! In the true spirit of Galaga, you will now be implementing the laser canon of your spaceship.

As with the `Enemy` class, create a `PlayerShot.cs` file with a `PlayerShot` class. Again, this should also have an identical constructor to the `Player` class. Inside the constructor, however, you should make an addition. Set the direction of the shot to $(0.0f, 0.01f)$ - this will give it a reasonable, constant speed upwards toward the enemies.

When you created the Enemies, you put them in a list, and we will be doing the same for the player projectiles, so add a list to the `Game` class called `playerShots` and instantiate it in the game constructor. This will be "holding" all the shots. *Be sure to make this list publicly readable and privately writable.*

Since it is the `player` object that is shooting, the `player` should also be responsible for creating the shots. Add the following method to the player class:

Input: None.

Output: `void`.

Behavior: Adds a shot to the `playerShots` list in the `Game` object reference from the constructor. When creating a `PlayerShot` instance, set the extent of your shot to $(0.008f, 0.027f)$; this gives the projectile a reasonably sized *hitbox*. Also, when adding a shot to the projectile list, add it along with the image `BulletRed2.png` and make sure its position is centered in front of the spaceship (that is the only logical place to put a laser cannon).
Hint: Load the image once, instead of every time a shot is added.

In this version of Galaga the spaceship should shoot its lasers whenever the player *presses* the `SPACE` button.

3.5.1 Adding Movement to the Player Shot

Since the laser shots need to interact with the enemies, we are going to make a function that can handle the updating and collision of the shots, rather than doing it directly in the `GameLoop`.

Define a function called `IterateShots` to iterate over all shots and their collision with the game's enemy entities. Beneath we show a sample implementation to get you started:

```
public void IterateShots() {
    foreach (var shot in playerShots) {
        shot.Shape.Move();
        if (shot.Shape.Position.Y > 1.0f) {
            shot.DeleteEntity();
        }
        foreach (var enemy in enemies) {
            // TODO: perform collision detection
            // (hint: Physics.CollisionDetection.Aabb)
        }
    }
}
```

Your function should implement the following strategy:

3.1. Move the shot using `shot.Shape.Move`.

3.2. If the shot "leaves" the coordinate system (i.e. its Y-position is greater than 1.0), it should be **marked for deletion using** `DeleteEntity`, as we will no longer need to move or render it. This is shown above.

3.3. If the shot has collided with an Enemy entity, we should **mark** both the Enemy and the shot **for deletion** itself from their respective containers. Implement this by iterating through the enemies, and check for each if they have collided with the shot using the `Aabb` function, which can be found in `DIKUArcade.Physics.CollisionDetection`.

   The collision detection algorithm (`Aabb`) returns a `CollisionData` object.
   Hint: look at the file for how to use this return object.

You may be wondering why we are using `DeleteEntity` to mark objects for deletion, rather than removing them directly from the list with e.g. `RemoveAt`. The reason for this is that *lists cannot be modified while they are iterated over*. This means that you will have to add two loops to the `IterateShots` list, that each generate a new list for both the `enemies` and `PlayerShots` list without the marked entities. Below is a suggestion for how to implement one of these loops:

```
List<Enemy> newEnemies = new List<Enemy>();
foreach (Enemy enemy in enemies) {
    if (!enemy.IsDeleted()) {
        newEnemies.Add(enemy);
    }
}
enemies = newEnemies;
```

Finally, you will need to add the rendering of the shots to the game loop, which you should be able to do by yourself.

3.6 Explosions

For this implementation, you will also be adding violent death scenes for the aliens - explosions! First, we need to load the image strides for the explosion animation - you have seen how to do this before.

We introduce two key components in `DIKUArcade` for creating animations (both located in the namespace `DIKUArcade.Graphics`):

`Animation` **-** A class to create an animation object, which given a specified amount of animation time (in milliseconds), a shape (position and size on screen), and an image stride for the graphics.

`AnimationContainer` **-** A class which functions as a wrapper for the `Animation` class. You will not need to manually create `Animation` objects, as this container will do it for you.

Now you should create the container and the strides to be used for the animations:

```
private List<Image> explosionStrides;
private AnimationContainer explosions;

public Game() {
    ...
    explosionStrides = ImageStride.CreateStrides(8,
        Path.Combine("Assets", "Images", "Explosion.png"));
    explosions = new AnimationContainer(<some_integer>);
}
```

The integer value for the constructor of the `AnimationContainer` should be a number big enough so that the container can contain a fitting number of animations at the same time. Consider this value.

Then, we need some standardized way of adding an animation to this container. This method, which we introduce below, could be called when you *iterate the player shots and find a collision*.

```
private int explosionLength = 500;

public void AddExplosion(float posX, float posY,
    float extentX, float extentY) {
    explosions.AddAnimation(
        new StationaryShape(posX, posY, extentX, extentY), explosionLength,
        new ImageStride(explosionLength / 8, explosionStrides));
}
```

Now, to call this method during iteration of the player shots, we need to find the position and extent of each animation.
Hint: Find these from the collided enemy object.

3.7 Score

To make the game slightly more competitive, for this last part of your implementation, you will add a score.

Create a new file `Score.cs`, add a class called `Score`, and add the following code:

```
private int score;
private Text display;

public Score(Vec2F position, Vec2F extent) {
       score = 0;
       display = new Text(score.ToString(), position, extent);
}

public void AddPoint(...) {
       ...
}

public void RenderScore() {
       display.SetText(string.Format("Score: {0}", score.ToString()));
       display.SetColor(new Vec3I(255, 0, 0));
       display.RenderText();
}
```

Add all needed references and fill out the missing implementation in your new class. In order to get our new score to work, we need to update the `Game` class. You will need to:

- Create a new instance of `Score` in the `Game` class and initialise it.
- Figure out when you should invoke the AddPoint method.
- Figure out when you should render the score using the RenderScore method.

3.8 Aftermath

If you have successfully implemented all of the above section, you should now have a basic but enjoyable state of `Galaga` .

This is no small feat, as we have introduced a lot of concepts and game mechanics. If you did not understand all parts of this assignment, we recommend that you read it through a couple of times (and browse through the `DIKUArcade` source code as well), as this game engine will be used throughout the course.

The only thing that remains now is to write a small report to document your work, and to answer some questions to solidify that you have gained a basic understanding of how to use our game engine.

4 Report

For this group exercise a short report must be written to document the work you have done, and to explain any important design decisions that you made along the way. You should structure the report like a technical report (similarly to the previous assignment), *but you should only include sections that you find relevant.* **The report is expected to be 2-3 pages, and must not exceed 3 pages.**

4.1 Questions concerning `DIKUArcade`

In this report all of the following stated questions must be answered in a way which convincingly states that you have an overall understanding of how to operate the `DIKUArcade` API.

- Try to explain the purpose of using the `DIKUArcade.GameEventBus` class.
- Explain how the unique behaviors of `DynamicShape` and `StationaryShape` can be accessed from the `Shape` field of an `Entity` object.
- Explain the purpose and difference between the `DIKUArcade.Graphics.ImageStride` and `DIKUArcade.Graphics.Image` classes.

4.2 Questions concerning you implementation of `Galaga`

Questions concerning your own implementation of `Galaga` must also be answered. This is a part of the software documentation process, which we will require that you do in much greater detail later in this course. For now, we will keep it around basic questions:

- How does your implementation make sure that logic and rendering updates are consistent across machines?
- How you limit the number of image reads from the harddrive when instantiating game entities?
- Describe how you have used object-oriented principles in your implementation.
- Did you implement some parts of your game differently from what this exercise paper suggested? If so, explain these differences and why you made them.

4.3 Link to repository

You **must** provide a link to your repository in this report. The front page / introduction is a good place for such a link. If you fail to provide such a link, then your TA cannot grade your hand-in.