CS4303 W11 Practical
1500157521
30th November 2018

Overview

The game I decided to implement is called Everything Does Not Revolve Around You and it's a logical 2D game with fixed pre-drawn levels and simple mechanic that involves object movement around player object's axis. The player is a character in the game that can move left or right and the hero can either be turned left or right. There are certain objects (dynamic objects) in the world that revolve around the hero by 180 degrees on the hero's y axis. The player turns by pressing Space and when the hero turns, all of the dynamic objects are turned at once. The catch is, that if one of the dynamic objects is in collision with the environment, all of the dynamic objects become transparent, lose their colliders, and the hero cannot change his distance towards them as they move with him. This is a simple mechanic that can on itself create an interesting game, however, I decided to add other mechanics such as area effectors, temporary object generators and detachable objects to showcase the possibilities of the game and so the current game should be taken as a tech demo rather than a full game.

Context

My game shares som superficial similarities with the game called *Echochrome*. I wasn't aware of the game when I came up with the idea, but the feedback to P3 lead me to it. Both my game and *Echochrome* share an idea of revolvement around an axis or a point. However, *Echochrome* is a 3D game where the point of revolvement is the centre of structure the hero walks on, whereas mine is a 2D game where the point of revolvement is the player's y axis and it doesn't turn around the whole world, but rather only some objects attached to the upper platform hanging above the player.

Execution

Revolve.jar file is in out/artifacts/Roguelike_jar folder and can be run by the

command:

```
java -jar Revolve.jar
```

In the case of wrong version, reset version in intellij and build to jar again.

Design and Implementation

Game Object

Each game object in the game is represented by the *GameObject* class. Game objects implement Component or Strategy design pattern which means, that each beahviour of a game object, such as movement, collision or turnability is stored in a separate class and referenced in the game object as a field. There are certain shared variables, such as position or velocity which can be accessed by any component but the main goal is to decouple the behaviour from the object and deal with each one separately. This means, that with right interfaces, a component can be changed without any change to the base object. For example, we can change box collider for a circle collider and no change in the GameObject class is required. The disadvantages include a slight performance loss when accessing the components and greater robustness of the code.

Components

Components include any behavior that influences an object's position, velocity, orientation, other game objects or other components of the game object.
For colliders, we can decide between circle and box colliders which are initialized by slightly different parameters. Colliders then detect collision with other objects and change the velocity of the game object accordingly. When touching, however, there's a certain limit to velocity of an game object, after which the object goes through another object because the collision wasn't even detected. This behavior is averted by lowering the gravitation vector and adding a slight bouncing effect to border collision.

The dynamic objects have to include *Turnable* component and the player has to include *Turn* component. These 2 classes deal with revolving and moving behavior of the dynamic objects when attached to the player. To attach any object with a collider to the player, we simply need a reference to the *Turn* behaviour of the player and the new dynamic object can be added by *addTurnables()* method. The game object to add does not even have to include the *Turnable* behavior, as if it doesn't, the behavior is added in *addTurnables()*. The simple initialization of the relationship between a dynamic object and the player object allows for simple implementation of attachable/detachable runtime behavior of different dynamic objects. *Attachable* component simply extends *Turnable* behavior and checks the mouse position, as the attachment or detachment of a dynamic object can be done by mouse clicking.

Other important component is *Renderer* which is further extended by *RendererComposite* and *RendererLeaf.* The *Renderer* class handles rendering of images onto the screen and setting their orientation and position difference toward the game object they're added to. *RendererLeaf* includes a single image rendered only once, whereas *RendererComposite* includes a set of the images from the same source. The composite behavior is used when initializing platforms with multiple image tiles.

Other components include

- *AreaEffector:* adds a *BoxCollider* to a game object and adds velocity to each object in the collider
- *Exit*: adds a *BoxCollider* to a game object and when the player object reaches it, a new level is loaded
- *Link:* handles velocity of link objects that create a rope hanging from the upper platform down to a dynamic object it attaches to
- *ObjectGenerator*: regularly creates copies of an object, and destroys them after given time
- *Temporary*: destroys a game object after a certain number of frames has passed

- *TurnPlatform*: created when *Turn* object is created on the player and handles movement, turning and stretching of the upper platform

Game manager

Game manager actually updates the game enviroment and UI and loads a new level each time the exit is reached. As there is only one Game manager per game, we can use Singleton design pattern on it. The Game manager class also inherits from the *PApplet* class, and therefore is the starting point of the application. Game manager is storing point of all the behaviour governing the game. The other alternative would be to store it in the game objects themselves, which would be impractical and too robust.

UI

The UI includes start menu UI with options to start from the tutorial, start from any level or exit and a game UI which is displayed during any level and involves buttons to restart the current level, go to menu and get a hint which is original for each level. Furthermore, labels for player controls are always displayed in the lower left corner. The buttons are highlighted when mouse is over them. The UI is implemented in *GameUI,* all of the *Label* objects are created here and all of the behavior involving clicking on them is implemented in the class as well. The UI functions as a component in that it can changed for another with the same interface, however, it is not referenced in a game object but rather in Game manager as it depends on many rules of the game and Game manager is where such behaviour is stored and so easily accessed.

Physics

The objects have a velocity which is stored as a vector and therefore displays both power and direction of a force. These vectors can then be added to or subtracted from each other and we have an easy way to use forces that affect a game object. Any object can be subject to gravity. The gravity is implemented as a constant acceleration of velocity in the direction to the ground. To slow this acceleration,we use a drag, which mulitplies any resulting velocity by a constant between 0 and 1. Gravity has to be added to velocity before all the other components of the game object are taken into account, as these components may include *Collider*, which needs to react to current velocity rather than the other way around.

Camera

The camera follows the player movemet, however, the following is not absolute. Rather there is an invisible square window inside the centre of the screen window that when crossed by the player, gives a signal to camera to start following the player movement. In practice this means that when the player is moving in the centre of the screen or close to it, the camera is still, however, when the player is getting closer to a corner of the screen, camera start following him. All of this behavior is implemented in *Camera* class.

Level Factory

The best part of my implementation, in my opinion, is the the easy creation of new levels and behaviours. This is mostly given by the use of Component design pattern that decouples a behaviour from the game object. All this comes into play in *LevelFactory* class that contains methods to create new objects with certain kind of behaviour specified by the parameters. Each level is represented as a method, where the game objects are initialized, and after reaching the exit, all of the objects are destroyed and a new method containing a next level is called. Metods such as *buildCollidable()*, *buildAttachable()* and *buildExit()* attach all the necessary components to each object, and the details for each component, which are often reused in different components for the same object, are specified by the user in parameters.

The levels are implemeted so that there is certain learning curve. The tutorial introduces player to all necessary components and then each level starts with just one, which is then extended by another one and so on. For example level 3 and 4 introduce falling through dynamic objects when turning, and then in level 6 this concept is used at the start of the level with detachability, where the player now knows, that he/she needs to detach first and only then turn, otherwise he would fall through.

Evaluation

I tested the game by using 2 classmates who were my test subjects. Both played the game completely and gave me following feedback:

1) The game has a fairly original idea in the core
2) The UI could use some improvement
3) The number of levels is low
4) Collision could use some more work
5) The camer movement is fluid and enhances the experience
6) The hints are fairly helpful
7) UI in the menu and during the playing acts as it should
8) The revolving works as expected

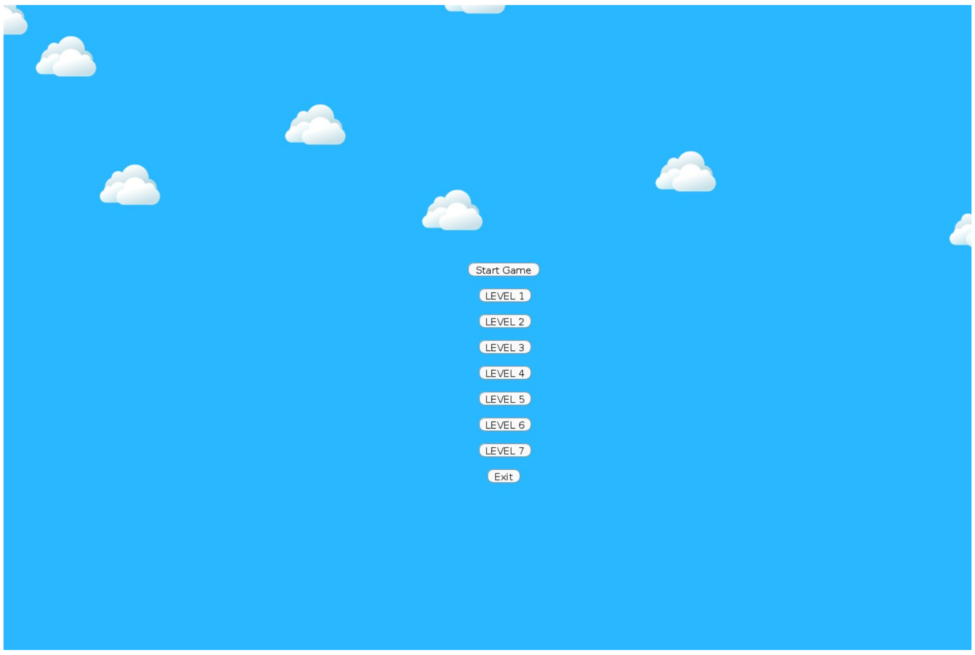
The implementations of the colliders are from previous practicals so they have already been tested by JUnit tests and work fairly well.

Conclusion

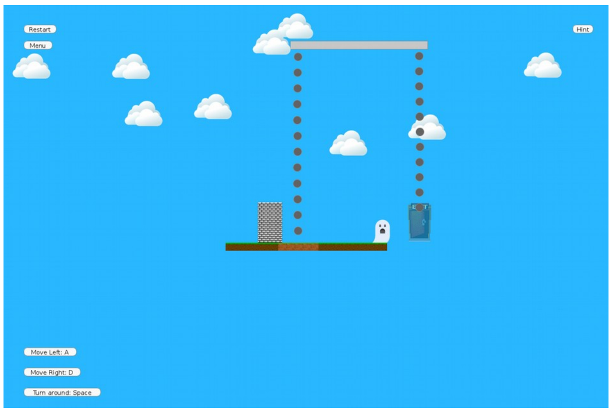
Overall, I implemented a working game with 7 levels and a tutorial that acts exactly as described in the lectures and is enhanced by other features such detachable objects, area effectors and object generators. The UI could use some more work but the actual game plays relatively well and is easy to extend for additional levels thanks to the *LevelFactory* class. The implementation uses Component design pattern heavily and is therefore also easy to extend for additional behaviour. The game has a moveable camera that loosely follows the player and saves some performance power thanks to that. What I found most difficult about the practical was to deal with the colliders as they still cause me some troubles in the code. Given more time, I'd do more testing and implemented more levels.

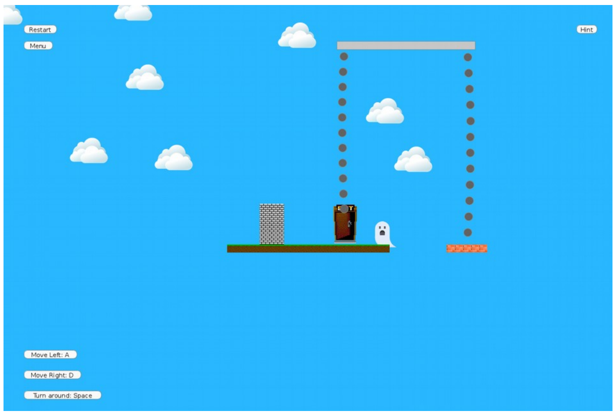
Testing

Game menu with options to start game from the tutorial, start from any level or exit the game.

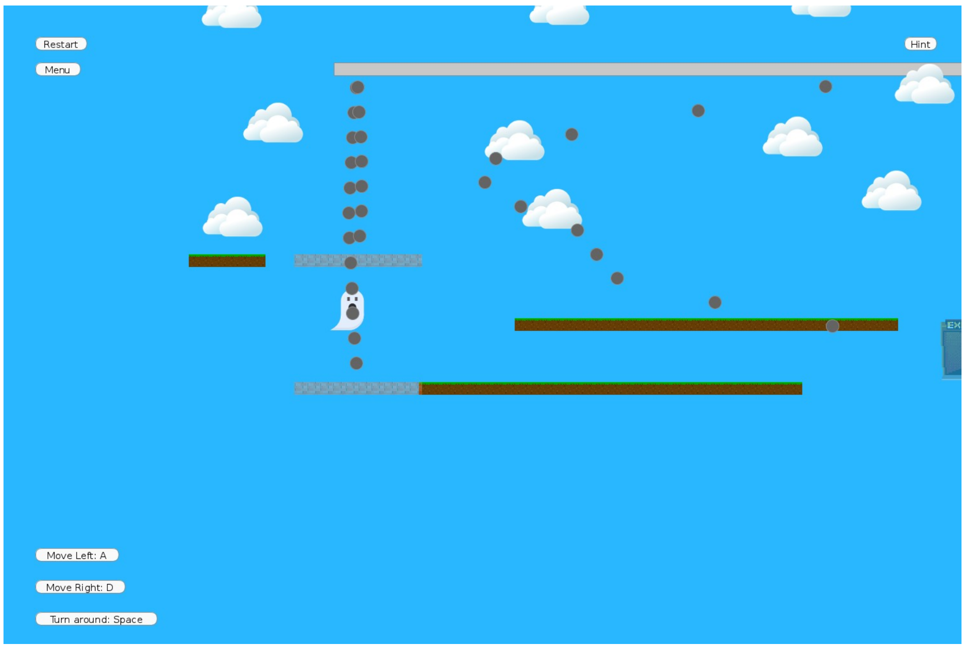


Before and after turn

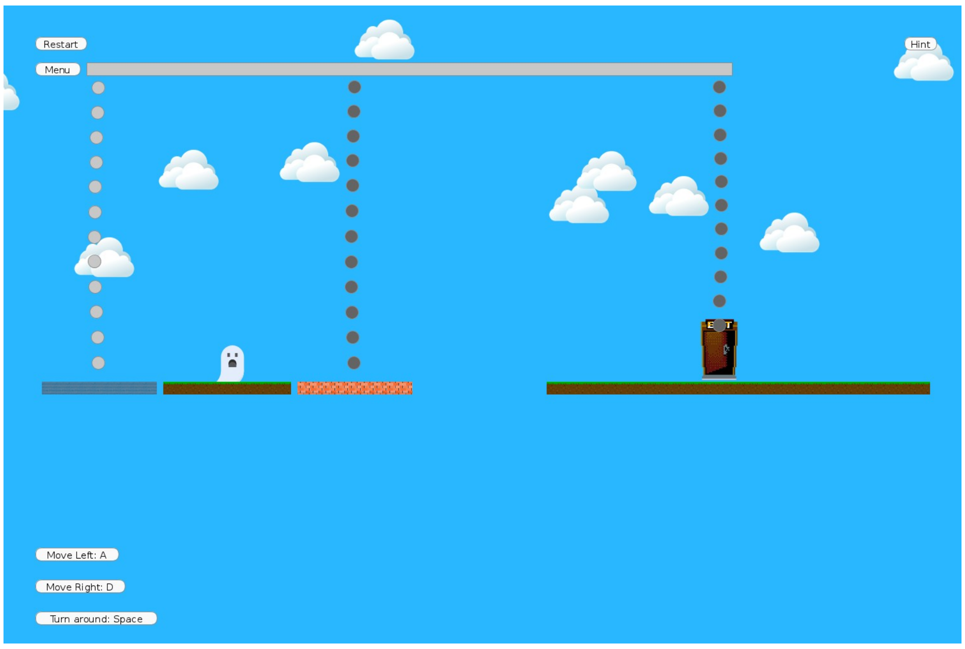




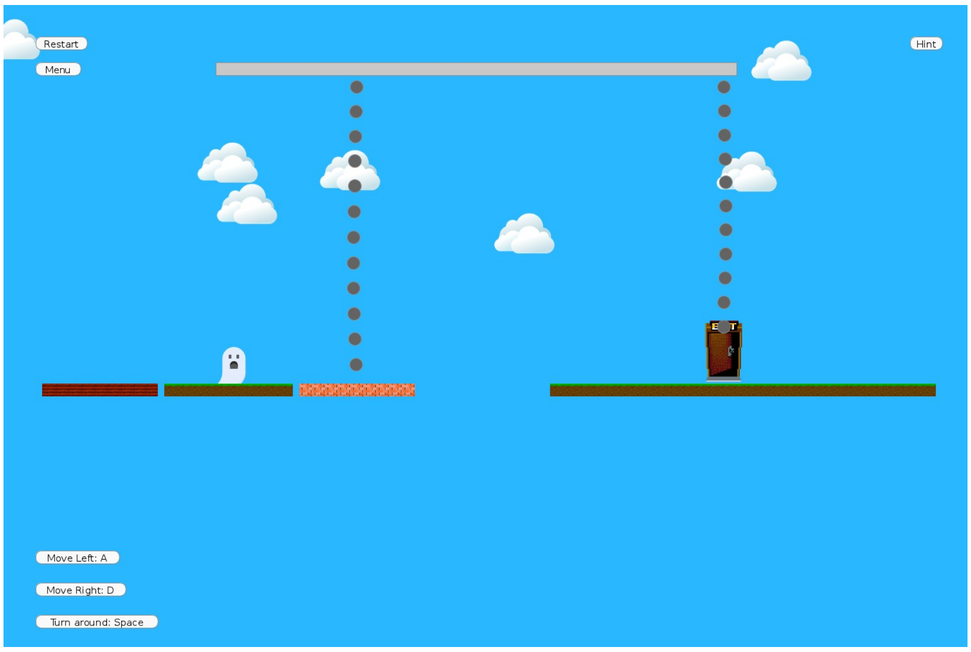
Not colliding with objects when they are transparent



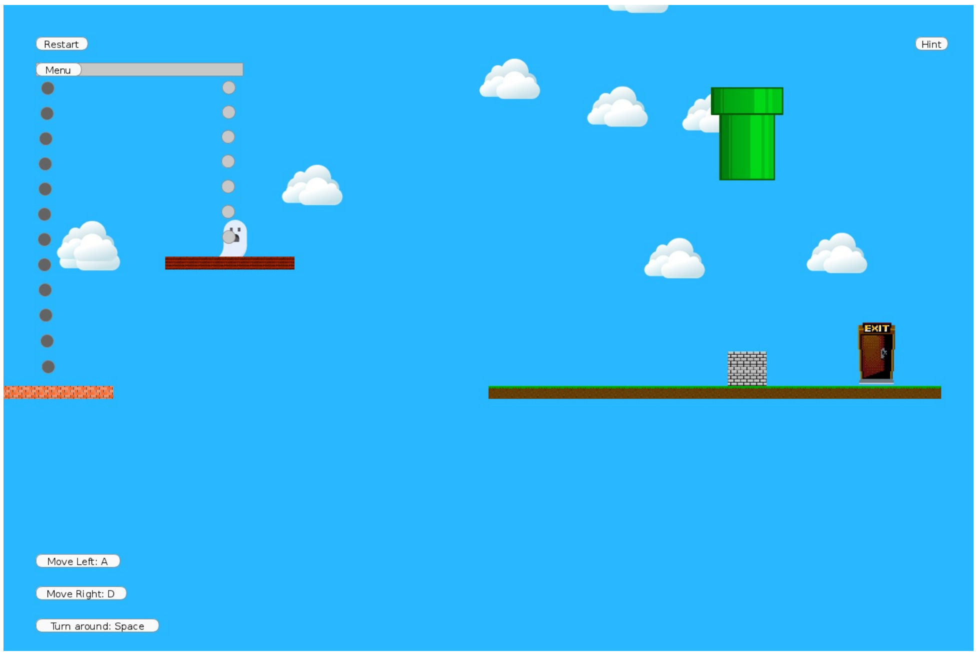
Detachable objects are highlighted when the mouse is over them



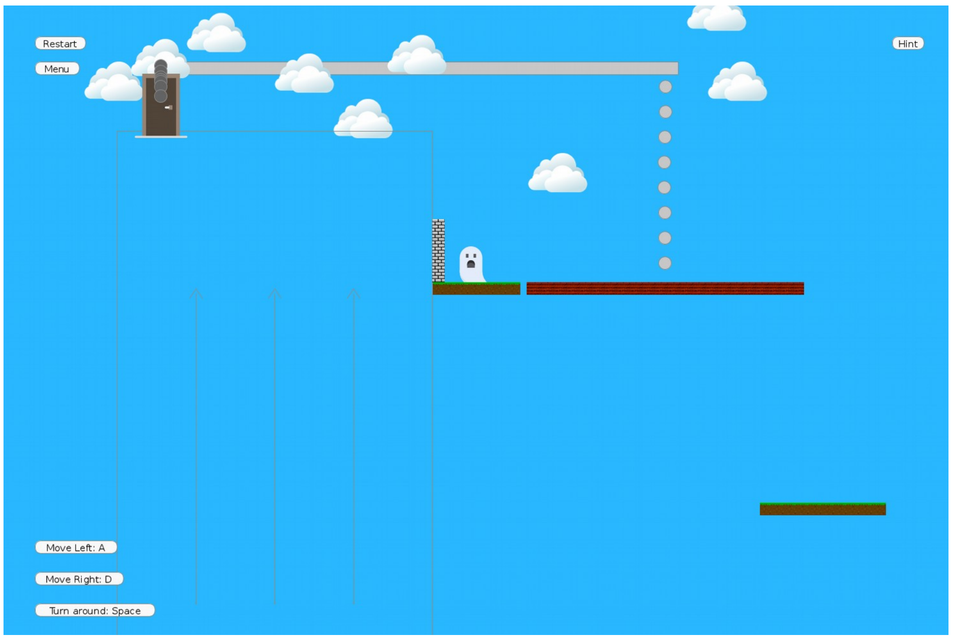
When clicked on, the object is detached, and the length of the platform is adjusted



Generator constantly creates new object that are destroyed after a time



Exit, which is dynamic in this case is afflicted by the area effector


Restart
Menu
Hint
Move Left: A
Move Right: D
Turn around: Space