1 Introduction

Having many different choices for the type of our project, choosing to make a computer game was never tough decision. Our belief was that making a game would be the most amusing and educative project type because the challenge of programming a computer game was something neither of use had ever faced before.

In the beginning of the process the ambition level was very high. Our intention was that the game should have network support, and even before the design phase, we sketched on the network protocol and implementation. It soon became clear that implementing the network support and all the other things we wanted was just too much work in too little time. But even though we narrowed down the list of game features it was obvious that the task wouldn't be a walk in the park.

Our idea isn't entirely made from scratch. There are some games in our past that has affected us and we have used elements we thought was positive from other games, and tried to avoid things we do not like in other games. The basic idea was inspired by an old Commodore 64 game called Space Taxi. There are however vast differences between our game and that. Similarities can also be drawn to games such as the classic Lunar Lander, and Sega® Crazy Taxi™.

2 Description of the game idea

The game takes places in a distant future. Mankind has colonized space, and research stations are spread out the universe to unravel the mysteries of space. It was immediately seen that there were problems transporting people between research stations. It wasn't economically viable to have a dedicated ship to each research station. This is where the player comes in. He takes the role as one of many interstellar taxi drivers, transporting people to remote outposts in deep space. It is not without competition though; other cabbies will be fighting for the few available customers.

The player will pilot a SX-NG 757 Deep Space Taxi Shuttle, or DSTS, which is equipped with a Class 5 shield to protect it from the dangers of space. Hitting another taxi or bumping into a platform at to high a velocity will activate its shield. Thrusting and having to use the shield will consume energy, and running out of it will lead to an almost certain death. If the player runs out of energy and his ship is destroyed, one can if there is enough credits available, purchase another ship. Credits will be earned from completing missions, i.e. transporting scientists to their destination platform. The winner will be the one to first reach the credits goal, or the last living player.

3 Initial work

The fact that we had such a high level of ambition in the beginning resulted in that we wanted many thing to be included in the game. Also being only three persons in the

group limited the size of the project. The must significant game features were (also included evident tasks):

- Be able to draw the game on a screen
- Collision detection between ships, platforms and so on
- Controlling the ships from the keyboard
- Music and sound effects
- Keeping track of score

The things we put up on the optional list are:

- Network support
- More sounds
- Fancy graphics
- Score board with many stats

After having spoken to our supervisor, Einar Pehrson, he had many suggestions on the features. He told us to among leave out network support optional, which we did, and replaced it with the possibility of having two players playing on the same keyboard. Looking back on project now, we can be content with Pehrson's advice.

Since we wanted to learn about the very foundations of game programming we left out the, in our opinion, bloated Swing package. This resulted in us among others having to implement our own double buffering.

4 Research

A fast paced action game is many ways much harder to program than for instance a board game or a turned based game. The main character being a space shuttle has to be very agile and maneuverable, and there will be a lot of different cases of collisions with other objects of the game. Collision handling is essential to the game, and to gain information about this matter we read tutorials on the web.

Another concern we had was implementing double buffering since a standard Java AWT component causes the screen to flicker when it's updated fast. A lot of info regarding different techniques, such as blitting, page flipping etc, was found on the web. The concept of sprites was also looked into; since it appeared that it would suite our project nicely.

A summary of books and web sites we've gotten information from can be found at the end of this document.

5 System parts and how they work

The system was divided up into a few main parts. A data repository was created to cache data for the whole game. For sound effects and music an audio player was needed, and together with the data repository, it is initiated during the loading phase of the game, at the splash screen module. The splash screen hands over control to a game controller, which is always present and controls most general aspects of the whole game, including menus and screen selection. It initiates and commands the screen manager, gets input from the keyboard manager and starts the game engine when the player wants to start a new game. The game engine is in charge of keeping the game running and handling events in the actual game play. These events include collisions, movement and other physics related tasks. Objects in the game space are called sprites, and have the ability to check for collisions, rotate, draw and animate themselves. For a reference of all classes see appendix C.

6 Dividing up the work

Being able to work towards an almost complete object orientation it was very simple to divide the project workload into parts that was developed separately. Each member of the group had his own area of interest which he was in charge of. Even though we divided the work, we still consulted each other if anyone had a problem or just a question.

7 Work methods

As mentioned earlier, since we were interested in making the game good, we early began sketching and thinking about how the game should work. Doing the requirement specification was pretty straightforward since we had the design pretty much drawn up in our heads and all we had to do was to get it on paper. However, later on we realized that the work would have been much more efficient if our design document would have been more detailed on the implementation aspects of the game. We had a few long discussions on how to do things and implement functions, something that could have been avoided with a more detailed design document.

Our programming strategy has been to continuously create objects along the way, grouping them into packages and putting them together as they were completed. We believe that this is a successful method of doing a computer game. At least it was for us.

8 Obstacles along the way

During the development we had a few obstacles that we needed to overcome.

8.1 Collisions

When playing the game you can collide with many different game objects (see the class GameObject in Appendix C). To detect collisions was not a problem, but how to calculate what should happen after they occurred was a serious issue. We solved this problem by picking up our old mathematics books and reading about some physics and simple vector mathematics, which we also applied to the collision handling.

8.2 Screen flicker

For drawing we use a buffered canvas which would prevent any screen flickering. When we begun the implementation phase we didn't have any real knowledge about how this was going to be done, but we had some ideas. After searching the net for examples and various articles (about graphics programming using Java) we succeed with the implementation of this system part.

8.3 Animation

Another problem we had problem with was animation and drawing the game. This was due to Java's way of repainting the screen. Our knowledge of Java's event thread wasn't deep enough, so we had to discover it by trial and error. We solved this putting the redrawing in an own thread, as should be done.

9 Final conclusions

We have learned a great deal about game programming, especially collision detection and handling and insight into Java™ graphics system. We are very pleased with what we have accomplished. However, we feel more effort should be put into the design phase. Something that should have been done was to complete more single classes before we started testing them and putting them together. Writing code to test every class and seeing how it worked took quite some time.

Altogether we feel we have learned a lot about game programming and game physics and we have enjoyed it all the way.

10 References

Fan, Joel (1996). *Black Art of Java Game Programming*. Sams

The Java Tutorial. (Electronic). Available: http://java.sun.com/docs/books/tutorial/.

Eric W. Weisstein. *Vector*. (Electronic). Available: http://mathworld.wolfram.com/Vector.html.

Patrick Niedermeyer & Joshua Peck (1997). *Exploring Java, 2nd Edition*. (Electronic). Available: http://skaiste.elekta.lt/Books/O'Reilly/Bookshelfs/books/java/exp/.

Joe van den Heuvel & Miles Jackson (1996). *Pool Hall Lessons: Fast, Accurate Collision Detection between Circles or Spheres*. (Electronic). Available: http://www.gamasutra.com/features/20020118/vandenhuevel_01.htm.

Paul Nettle (2000). *General Collision Detection for Games Using Ellipsoids*. (Electronic). Available: http://www.gamedev.net/reference/articles/article1026.asp.

Appendix A

Requirement Specification for a Computer Game

Working title: **SEDNA**

Introduction

The game is intended to be a type of Lunar Lander vs. Crazy Taxi. The objective is to transport passengers from one platform to another in a space ship (taxi). Ships and platforms are entities and can collide, which causes the ship to lose energy and forcing the player to refuel at the "gas station" or, if he fails to do so, dropping outside the screen due to gravity, and thus dieing. Obviously, an important objective is to keep the ship flying for as long as possible, but strategically performed collisions can also be winning tactic (and extremely annoying for the opponent).

1. **Scope**
   Taxi simulation in a space environment.

2. **Product Functions**
   Basic features of the game play is to control a space ship, picking up and transporting passengers to their destination and refuel the ship.

Specific Requirements

User Interface Requirements

The interface presented to the user should consist of a number of easily understood *screens*. The first screen a *title screen* where the user can select one of the following options

1. Play
2. Instructions
3. Credits
4. Exit

Except for the *Play screen*, all others are considered self-explanatory.

The *play* or *game screen* is where the actual playing is performed might look something like this:

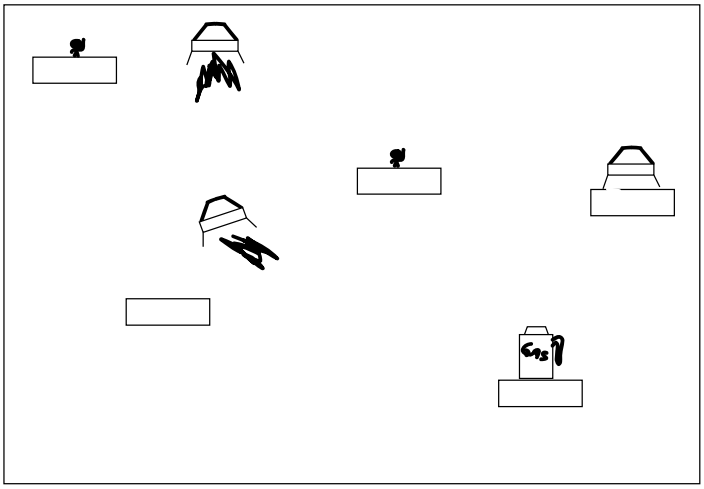


Figure 1, a simple illustration of the game screen.

Scores, fuel and additional indicators will be displayed somewhere where it doesn't affect game play, i.e. at the top of the screen. The screen will wrap on the horizontal axis and a player shouldn't be able to reach areas above the top and below the bottom of the screen. These areas are out of bounds which mean that the player needs to respawn.

After completed game a *score table screen* will be displayed.

Functional Requirements

This section covers the different parts of the game from a greater perspective than what the player actually sees but, nevertheless, they are of great importance.

The World

As said earlier, the game takes place in outer space. This automatically raises questions concerning movement capabilities, of course, how the environment should look, which is described in User Interface Requirements.

Physical Forces

The world should not impose any force on moving objects that makes them stop, e.g. friction. The outcome of this is that when a certain speed is reached the object should continue with the same speed if no "outer force" interferes. However, the world should have "gravitational" features which affect movable objects as an outer force, pulling ships towards the bottom of the screen.

Note that the definition of gravity is a force of attraction between two or more objects. The mass of ships, platforms or any other (small) objects in the world are considered too small to affect each other.

The Objects

In the game there should be two basic types of objects: moveable actors and inert objects that are called *in*actors. Actors are ships, passengers and more … such as explosions.

Ships

The player controls a space ship and uses it to land on platforms, pick up a passenger and transport the passenger to its desired destination. The reward for a completed mission should be score in forms of completed assignments and credits. The credits should be used for refueling the ship (see Platforms).

Inputs

A ship is controlled by (arrow) keys but exceptional events such as object collisions requires ships to be controlled by the game model.

Processing

Apart from the steering, the ships should be able to collide with other actors or inactors. Collision will cause actors to deflect, but inert objects should maintain their position. For this to be possible a collision detection mechanism is needed as well as a quick way to calculate new velocities and directions on the colliding objects. A collision activates a shield around the ship, which uses up some energy.

Output

Information stored as specific ship data should be accessible from the outside world. The variables are position, direction, speed, energy, size/bounds, scores and looks. Outside "interference" on ship data must somehow be restricted.

Platforms

Objects that aren't able to move are considered stationary, or actually platforms since no ships are stationary. Platforms should be landing sites for the ships and but also host/destination for every passenger. A special case of a platform is the *refuel station/gas station* where energy can be bought. This platform should have extra features such as giving energy to a ship in exchange for credits.

Inputs

Passenger platforms should be alerted when new passengers spawn and when ships land. The latter should also apply to the refuel platform.

Processing

When a player lands his/her ship on a regular platform and a passenger is present, then the passenger should automatically board the ship and the destination should be indicated. The platform should act as a messenger between the passenger (somehow spawned by the platform itself) and the ship and should keep track of ships that are present at the platform and also available to pick up a passenger.

Output

The platforms will spawn passengers.

Passenger

Passengers are added to make the game competitive. The main objective is to move passengers from their spawn platform to their desired destination platform (randomized) and score by completing a route. All passengers are more or less static and should not require any input, output or processing.

The Game Itself and How to Win It

The game model must enforce some rules. A few of them are mentioned above as a functional requirement, but many others, often essential to the player, should have more specific regulations and you can not be too clear.

Scoring

Rewards for completing an assignment (transporting a passenger from its start to its destination) are given in form of simple points, one for each assignment, but also in form of space money or credits. The credits should be used by the player to buy energy to his/her ship. Energy is fuel and without fuel, the ship won't fly.

Staying Alive

Even though the game isn't about dieing, a player that ends up below the lower part of the screen is out of bounds and will have to respawn. Sinking too low can be the outcome of many situations. One can be pushed down or simply lose energy, making the ship unable to travel upwards. Scores should not be affected by a respawn; however the player loses a noticeable amount of time and will have to buy fuel for a respawn. There should also be a fixed fee for a respawn. This should mean that if a player is low on credits and therefore cannot afford the respawn fee plus some additional fuel, the player is out of the game. Colliding with an object when you are out of energy will make your ship implode. Also, if a player runs out of energy above a platform, and manages to land, this will count as a crash. When trying to land on a platform, the users approach should be fairly vertical and the speed should not be too high. Anything else will result in a bounce off and shield drain.

Winning the Game

A player will strive to makes as many transports as possible. The game should be finished when a player has made completed a certain amount of assignments or has made a certain amount of credits. The game should also end if a player travels out of bounds and does not have sufficient credits to re-enter the game (see Staying Alive).

Performance Requirements

Since the project is a fast-paced game its reaction must be instant and very accurate. One thing that must be taken into consideration is the rather slow process of creating objects

in Java. We will strive to keep new object spawning to a minimum, while maintaining a fully object oriented approach.

Optional Features

There are a few optional features that we want to include, but the time schedule sets limits to what can be accomplished. The features are ordered in the way we think they should be implemented with the one we like best on top.

1. **Network**
   Implement a network layer to act as a network session manager that keeps track of connecting/disconnecting players and updating game status. The network model should be client-server with stand-alone server software. More specifically the approach should use UDP socket for transferring packets since the game shouldn't require the secure functions of a TCP socket and it will also keep the traffic low. If there is time we'd also like to implement inter- and extrapolation for additional smoothness, especially if the networking core isn't "fast enough".

   This network feature will require some manipulation of the current menu system, most likely by adding options like "Host Game" and "Join Game". What they do is obvious.

2. **Sound**
   Music melodies and ingame sound effects will certainly enhance the game experience, but are not crucial.

3. **Fancy Graphics**
   There will be really neat graphics if there is time.

4. **Extended Scoreboard**
   When the game ends, game screen will freeze, blur and the score table will be superimposed on it.

5. **Stats, stats, stats!**
   On the high score screen, user will be presented with lots of different useful and useless stats, e.g. number of collisions, number of crashes, amount energy refueled.

Note that the *Optional Features* require a rather large set of additional rules that aren't, widely discussed in the text.

Appendix B

Interstellar Taxi: User manual

Winning the Game

A player will strive to makes as many transports as possible. The game ends when a player has made completed a certain amount of assignments or has made a certain amount of credits. The game also ends if a player travels out of bounds and does not have sufficient credits to re-enter the game (see Staying Alive).

Controlling the ships

Player one uses the arrow keys.
Player number two uses W, A, S and D to go up, left, down and right (respectively).

Scoring

Rewards for completing an assignment (transporting a passenger from it's start to it’s destination) are given in form of simple points, one for each assignment, but also in form of space money or credits. The credits are used by the player to buy energy for his/her ship. Energy is fuel and without fuel, the ship won’t fly.

Staying Alive

Even though the game is not about dieing, a player that ends up below the lower part of the screen is out of bounds and will be respawned. Sinking too low can be the outcome of many situations. One can be pushed down or simply lose energy, making the ship unable to travel upwards. Scores isn’t affected by a respawn; however the player lose noticeable amount of time and will have to buy fuel for a respawn. There is also a fixed fee for the respawn. This means that if a player is low on credits and therefore cannot afford the respawn fee plus some additional fuel, the player is out of the game. Colliding with an object when you are out of energy will make your ship implode. Also, if a player runs out of energy above a platform, and manages to land, this will count as a crash. When trying to land on a platform, the users approach should be fairly vertical and the speed should not be too high. Anything else will result in a bounce off and shield drain.

Appendix C

Design Document

Sedna, Computer Game.

When the game is started, the user will be presented with a splash screen. Here various files will be buffered and among others, the sound subsystem will be initialized. When all caching is done, a menu screen will be shown. Here the user is presented with four choices: Start Game, Instructions, Credits and Exit. Instructions and credits will take the user to a static screen with some information regarding each subject, and an option to return to the main menu. Exit is considered self explanatory. Start game will display the actual play screen. When the game is finished, a choice will be given to either return to the main menu or play another game.

Classes

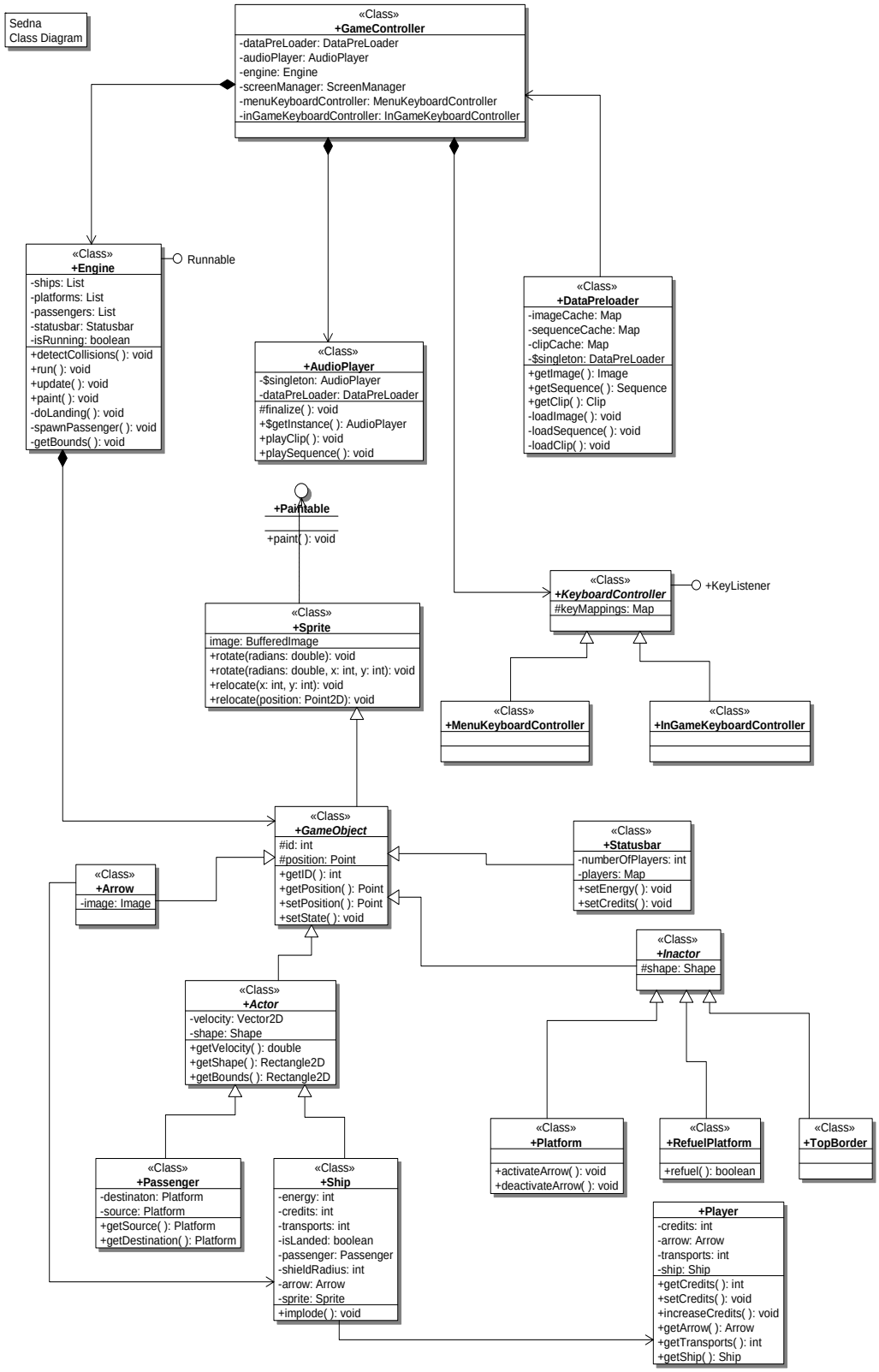
Class name	Description	Est. time of implementation (hours)
Actor	Abstract class to "actors" in the game field.	5
Arrow	Points out the passenger's destination.	3
AudioPlayer	Handles playing of sound effects and music.	3
BufferedCanvas	A dubble-buffered canvas.	8
CreditsScreen	Displays credits. Information about who created the game, etc.	3
DataPreloader	Caches all files that are to be used.	10
Engine	The game engine. Controls the various aspects of the game, such as drawings, updates, sprite collisions, etc.	25
GameController	Acts as a game supervisor.	15
GameObject	Abstract class to Actors, Inactor and Status. Simply all objects of the game field.	10
Inactor	Abstract class to "inactors" in the game field.	5

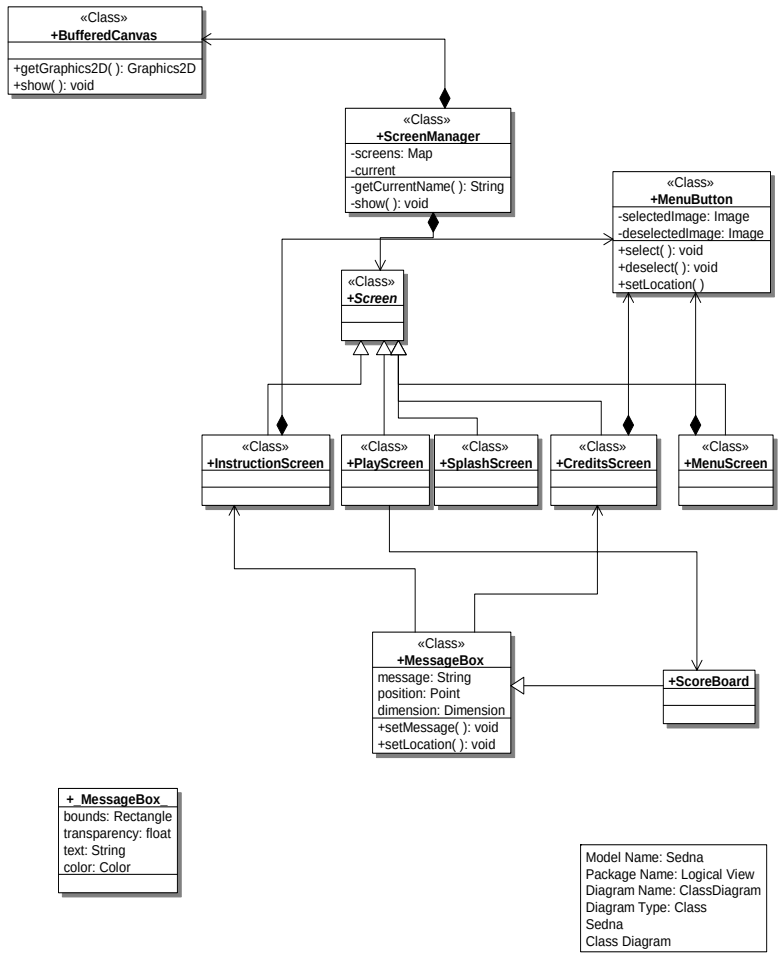
InGameKeyboardController	The In-game keyboard controller.	5
InstructionScreen	Shows instructions for how the game is played.	2
KeyboardController	Handles keyboard input	10
MenuButton	Used for displaying/selecting options in menus, etc.	3
MenuKeyboardController	Keyboard input for the menu screens	5
MenuScreen	Shows the title and various options.	5
MessageBox	Displays text messages in a bounding box.	5
Passenger	Displays the object to transport, keeps track of the source and destination.	3
Platform	Platforms where passengers will appear and be delivered to.	5
Player	Keeps track of each player and its ship.	1
PlayScreen	The screen where all the playing takes place.	5
RefuelPlatform	The platform where the ships land to refuel. Can refuel a ship.	5
ScoreBoard	Displays a sorted score board.	3
Screen	Abstract class for all screens.	5
ScreenManager	Handles all cards/screens. Works like cardlayout, but has the additional option to select a screen by name.	8
Ship	The ship object.	6
SplashScreen	Screen that shows when DataPreloader starts, before the menu is shown.	2
Sprite	Super class for all sprites. Has methods for rotating, collision checking etc.	10
TopBorder	Acts as a top boundary which the ships can collide with.	1

	TOTAL: 176

Note: the above numbers are a bit high and some classes which depend on each other will take considerably less time to code. However, there are always obstacles on the way which require extra time. Still, the total number of hours is a bit pessimistic.

Drawing graphics and finding music and soundeffects will probably take another 6-7 hours.


Sedna
Class Diagram
«Class»
+GameController
-dataPreLoader: DataPreLoader
-audioPlayer: AudioPlayer
-engine: Engine
-screenManager: ScreenManager
-menuKeyboardController: MenuKeyboardController
-inGameKeyboardController: InGameKeyboardController
«Class»
+Engine
-ships: List
-platforms: List
-passengers: List
-statusbar: Statusbar
-isRunning: boolean
+detectCollisions( ): void
+run( ): void
+update( ): void
+paint( ): void
-doLanding( ): void
-spawnPassenger( ): void
-getBounds( ): void
Runnable
«Class»
+AudioPlayer
-$singleton: AudioPlayer
-dataPreLoader: DataPreLoader
#finalize( ): void
+$getInstance( ): AudioPlayer
+playClip( ): void
+playSequence( ): void
«Class»
+DataPreloader
-imageCache: Map
-sequenceCache: Map
-clipCache: Map
-$singleton: DataPreLoader
+getImage( ): Image
+getSequence( ): Sequence
+getClip( ): Clip
-loadImage( ): void
-loadSequence( ): void
-loadClip( ): void
+Paintable
+paint( ): void
«Class»
+KeyboardController
#keyMappings: Map
+KeyListener
«Class»
+Sprite
image: BufferedImage
+rotate(radians: double): void
+rotate(radians: double, x: int, y: int): void
+relocate(x: int, y: int): void
+relocate(position: Point2D): void
«Class»
+MenuKeyboardController
«Class»
+InGameKeyboardController
«Class»
+GameObject
#id: int
#position: Point
+getID( ): int
+getPosition( ): Point
+setPosition( ): Point
+setState( ): void
«Class»
+Statusbar
-numberOfPlayers: int
-players: Map
+setEnergy( ): void
+setCredits( ): void
«Class»
+Arrow
-image: Image
«Class»
+Inactor
#shape: Shape
«Class»
+Actor
-velocity: Vector2D
-shape: Shape
+getVelocity( ): double
+getShape( ): Rectangle2D
+getBounds( ): Rectangle2D
«Class»
+Platform
+activateArrow( ): void
+deactivateArrow( ): void
«Class»
+RefuelPlatform
+refuel( ): boolean
«Class»
+TopBorder
«Class»
+Passenger
-destinaton: Platform
-source: Platform
+getSource( ): Platform
+getDestination( ): Platform
«Class»
+Ship
-energy: int
-credits: int
-transports: int
-isLanded: boolean
-passenger: Passenger
-shieldRadius: int
-arrow: Arrow
-sprite: Sprite
+implode( ): void
+Player
-credits: int
-arrow: Arrow
-transports: int
-ship: Ship
+getCredits( ): int
+setCredits( ): void
+increaseCredits( ): void
+getArrow( ): Arrow
+getTransports( ): int
+getShip( ): Ship


«Class»
+BufferedCanvas
+getGraphics2D( ): Graphics2D
+show( ): void
«Class»
+ScreenManager
-screens: Map
-current
-getCurrentName( ): String
-show( ): void
«Class»
+MenuButton
-selectedImage: Image
-deselectedImage: Image
+select( ): void
+deselect( ): void
+setLocation( )
«Class»
+Screen
«Class»
+InstructionScreen
«Class»
+PlayScreen
«Class»
+SplashScreen
«Class»
+CreditsScreen
«Class»
+MenuScreen
«Class»
+MessageBox
message: String
position: Point
dimension: Dimension
+setMessage( ): void
+setLocation( ): void
+ScoreBoard
+_MessageBox_
bounds: Rectangle
transparency: float
text: String
color: Color
Model Name: Sedna
Package Name: Logical View
Diagram Name: ClassDiagram
Diagram Type: Class
Sedna
Class Diagram