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.