Jurassic Ducks report

Ren Ding

u5111810@anu.edu.au

27^{th} October 2012

A report submitted for the degree of Master of Computing at
Australian National University

Supervisor: **Mr Hugh Fisher**

Course Code: **COMP8780 IHCC Project**

Acknowledgements

I'd like to thank my supervisor Hugh Fisher for his technical support and advice on this project. I'd also like to express my appreciation to the course coordinator Weifa Liang for teaching me research report writing skills. Finally, I thank my girlfriend Yuyu, my parents and my friends for their encouragement.

Content

Abstract

Jurassic Ducks is a 3D Game designing project. It contains many techniques, which are artificial intelligence algorithms design, photorealistic rendering and software engineering. The main activity of the game is that the player, who is trapped around ANU Computer Science and Information Technology (CSIT) Building by a flock of Jurassic ducks, tries to escape from these ducks. The program can also run on cross-platform operating system.

Keywords: 3D Game, AI, Computer Graphics, OpenGL, Software engineering, Cross-platform

1. Introduction

The 3D game project Jurassic Ducks is designed from scratch, and it is not based on any previous work before. This report will show techniques used in the project in terms of game AI algorithms design, graphics rendering and software development framework. It will also show what techniques and algorithms did not work in the project.

Jurassic Ducks is a First-Person view game, the user can change the player's view by mouse and control the player's movement by keyboard. There is also an optional key for the player to look at the game in the bird's-eye view. The player needs to evade from the computer-controlled ducks. Once enough ducks touch the player, the game is over.

The program was written in C++. The motivation of the choice of this programming language is that first I am familiar with this programming language; second, C++ is a compiled language, which is faster than many other languages such as the virtual machine language Java for the same structure code. Third, the combination of C++ and OpenGL is popular and commonly used in the graphics program. While developing the project, my supervisor Mr. Hugh Fisher offered me map loader codes in Python and GPU shade loader codes in C. Then, I wrote a C++ version based on these. Qt (Jasmin and Mark, 2008) is the GUI framework used in this project because it is a cross-platform (a project requirement) GUI framework that is widely used for developing application software. Also, Qt is better than GLUT (Dave, 2009) because it has an explicit frame loop and richer GUI options. Further, the program was mainly tested on Macintosh OS and Ubuntu OS.

Because of the limitation of my laptop Macbook pro 13', I chose OpenGL 2.X with GLSL1.2 (Richard et al., 2007) instead of a higher version OpenGL 3.X with GLSL 1.4. To do more photorealistic effects, the higher version OpenGL and GLSL would be helpful.

For the project version control, I used Git (git-scm.com) and Bitbucket (bitbucket.org) to manage the program versions. The project code can be downloaded from (https://bitbucket.org/RenDing/jurassicducks). To compile the program, Qt4, CMake, G++ are required. It also requires that the computer graphic card can support OpenGL 2.X and GLSL 1.2.

2. Artificial Intelligence in the Game

2.1 Chasing and Evading

This section describes which of the chasing and evading algorithms are applied in the Jurassic Ducks project and which were abandoned. It will also analyze the drawbacks of each algorithm and how to overcome these.

2.1.1 Basic Chasing and Evading

Although Jurassic Ducks is a 3D game using a right-handed coordinate system (Dave, 2007), the AI algorithms take place in 2D. The ducks and the player walk on the plane, which is made by x-axis and z-axis. Their position can be specified as (x, z), the positive x-axis points to the right, and the positive z-axis points to the viewer.

The simple and naïve chase and evading algorithm involve correcting the ducks' (predators') coordinates based on the player's (prey's) coordinates. The way of correcting is by reducing the distance between their positions. For example, one duck's position is (6.0, 4.0), the player's position is (1.0, 1.0). By this algorithm, the duck's chasing path is (6.0, 4.0) -> (5.0, 3.0) -> (4.0, 2.0) -> (3.0, 1.0) -> (2.0, 1.0) -> (1.0, 1.0).

Jurassic Ducks is a tile-based game, which uses discrete tiles as one of the fundamental elements of play. Therefore, ducks' and the player's positions are fixed to a discrete tile and specified as a tuple of integer values. To improve the algorithm, the only difference from the previous example is that rows and columns are used instead of floating-point value (6, 4) -> (5, 3) -> (4, 2) -> (3, 1) -> (2, 1) -> (1, 1).

The trouble with the basic chasing and evading algorithm is that ducks' chasing behavior often seems mechanical. Take Figure 2-1 as an example, the duck first moves diagonally toward the player. Once one of the coordinates, the vertical in this case, equals that of the player's, the duck advances toward the player straight along the other coordinate

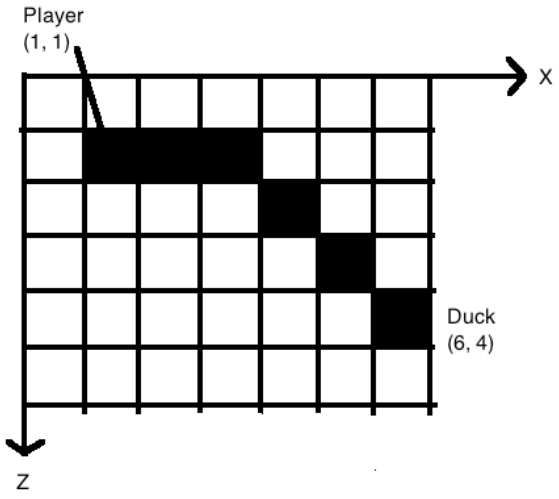

Figure 2-1: Basic chasing and evading

axis, the horizontal in this case. It looks so unnatural in this example. In jdalgorithm.cpp, the *basicChasingEvading* function implements this algorithm in the tiled environment.

2.1.2 Line-of-Sight Chasing

To improve the basic chasing and evading algorithm, a better approach is to have the duck move directly toward the player in a straight line. The main point of Line-of-Sight Chasing (David and Glenn, 2004) is that the predator always moves directly to the prey's current position. Figure 2-2 shows the comparison of simple chasing and Line-of-Sight chasing.

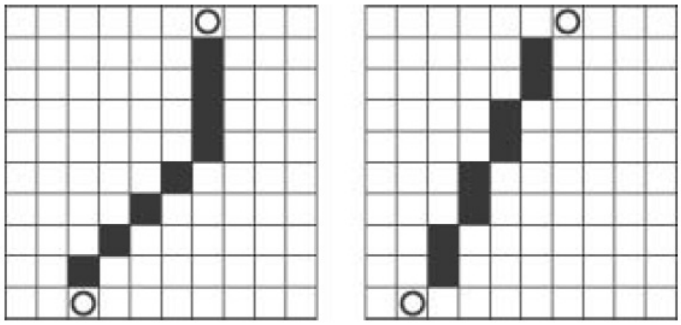

Figure 2-2: Simple chasing vs line of sight chasing (http://commons.wikimedia.org/wiki/File:Simple_chasing_vs_line_of_sight_chasing.jpg)

In the Jurassic Ducks project, two algorithms implement Line-of-Sight chasing. One applies to tiled environments, while the other is specifically for continuous environments (Andreas, 2009).

In the tiled environment, one of the big problems is that movement can appear jaggy. The reason is that screen pixels are typically small so that each tile is not mapped to a screen pixel. To solve this problem, ducks should only move to adjacent tiles when changing position. In the Jurassic Ducks game, this only offers eight possible directions of movement (Figure 2-3). However, mathematically speaking, none of those directions can accurately represent the true direction of the target. As you can see, none of the eight directions leads directly to the player. What should be done is a way to determine which of the eight adjacent tiles to move to so that the duck appears to be moving toward the player in a straight line.

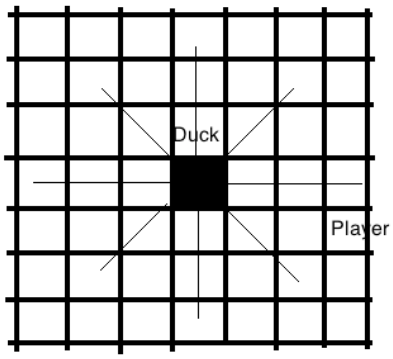

Figure 2-3: Tile-based eight-way

Bresenham's line algorithm (1956) is helpful for solving this problem. The Bresenham line algorithm is an algorithm which determines which points in an 2-dimensional pixel based environment should be plotted in order to form a close approximation to a straight-line between two points. The Figure 2-4 shows an example of the Bresenham line algorithm.

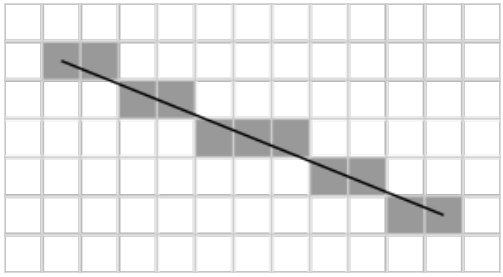

Figure 2-4: Bresenham line algorithm result
(http://en.wikipedia.org/w/index.php?title=File:Bresenham.svg&page=1)

The Equation 2-1 (Bresenham, 1956) represents the slope of the line. The key to drawing the line is adjusting the next point to keep the invariant condition (slope) to be always true.

$$S_{line} = \frac{Z1-Z0}{X1-X0} \qquad \text{Equation 2-1}$$

For more detail, please see jdalgorithm.cpp. The *lineOfSightTile* function implements the Line-of-Sight chasing algorithm, based on Bresenham's line algorithm, in a tiled environment.

In continuous environments, the predator's movement behavior is totally different. Applied forces and torques drive the predator. In the Jurassic Ducks project, each duck has two types of force, which are steering force and thrust force (Figure 2-5). In the chasing state, each duck will constantly calculate the player's position relative to it and adjusts its steering to keep itself pointed directly toward the player. In Figure 2-6, the duck has its own coordinates. When the player is at the duck's port side (left-side, positive x value), the duck can strengthen its left steering force to move left or vise versa (negative x value).

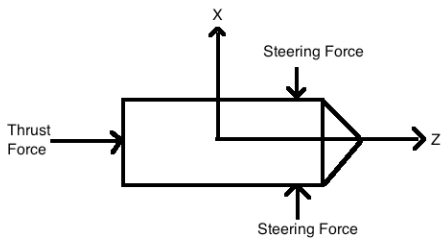

Figure 2-5: Duck forces

The main steps of steering force adjustment are:

1. Convert each duck's position from world coordinates to local coordinates;
2. In the duck's local coordinates, if the x position value is greater than 0, strengthen right steering force; if the x position value is less than 0, strengthen left steering force.

The Equation 2-2 (David and Glenn, 2004) is used to convert each duck's position from the fixed world coordinates to its local coordinates.

$$x = X\cos\theta + Z\sin\theta$$
$$z = -X\sin\theta + Z\cos\theta$$

Equation 2-2

Here, (x, z) are the local coordinates of the global point (X, Z). Figure 2-6 shows the system conversion. In jdalgorithm.cpp, the vRotateCoord function implements the equation 2-2.

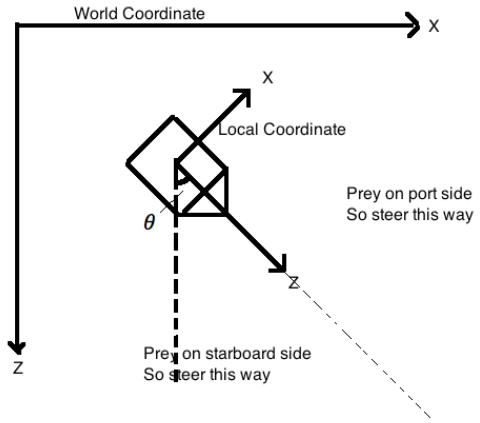

Figure 2-6: Global and local coordinate systems

2.1.3 Interception

The Line-of-Sight chase algorithm is more effective than the basic chasing. However, that algorithm still has drawbacks. One of them is that heading directly towards the

prey is not always the shortest path in terms of range to target (or time to target). Figure 2-7 clearly shows the drawback.

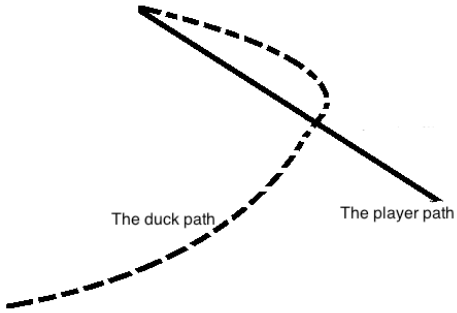

Figure 2-7: Line-Of-Sight chase in continuous environment

As the figure illustrates, the player started from the bottom right of the map and the duck started from the bottom left to chase the player. Over time, the player moved in a straight line toward the upper left corner. The duck's path curved as it continuously adjusted its heading to keep pointing to the moving player. The drawback is that the duck will always end up right behind the player, except it is moving so fast that it overshoots the player.

To overcome this shortcoming, the interception concept (David and Glenn, 2004) in games is introduced. The key is to find the interception point. Here, the interception point is specified as P_i, which is a vector.

$$P_i = P_{duck\ current} + V_{duck\ current} * t \qquad \text{Equation 2-3}$$

In Equation 2-3, $P_{duck\ current}$ is a vector, which represents the duck's current position. $V_{duck\ current}$ is also a vector, which represents the duck's current velocity. t is the time that the duck will take from current position to intercepting point.

The time is calculated as follows,

$$t = \frac{P_{player} - P_{duck}}{V_{player} - V_{duck}} \qquad \text{Equation 2-4}$$

It is equivalent to the relative distance between the player and the duck divided by the relative velocity between them.

Then, the Line-of-Sight algorithm is applied to it. Comparing to Line-of-Sight algorithm, the only difference in the Line-of-Sight intercepting algorithm is using the

predict point instead of the current duck position. The result is illustrated in Figure 2-8.

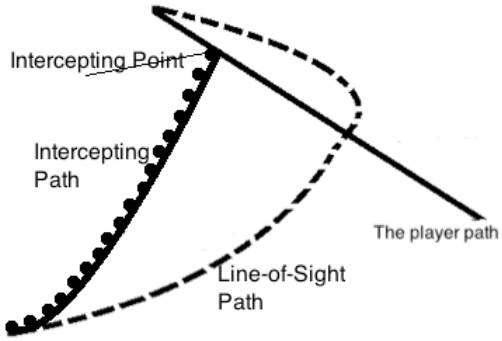

Figure 2-8: Line-Of-Sight Intercepting chase in continuous environment

Clearly, the intercepting algorithm yields a shorter path than the common Line-of-Sight algorithm. However, this algorithm has one constraint that the duck must move faster than the player, otherwise the t in Equation 2-4 will be infinity. Because the duck in the project move slower than the player, this algorithm was not used.

2.2 Pattern movement

This section will describe the Jurassic Ducks' pattern movement (David and Glenn, 2004) in tiled and continuous (physically simulated) environments respectively.

To make the ducks look more intelligent, pattern movement is a simple and good choice. The basic idea is that the ducks move according to some predefined pattern that makes them appear as if they are performing complex and thought-out movement. In the Jurassic Ducks project, some ducks do different types of patrolling such as square path, arbitrary path. Once the player moves close enough to these ducks, these ducks will start to chase the player. On the other hand, if the player moves out of the ducks' chasing area, these ducks will keep patrolling.

Technically, the main algorithm is to save the control data into set of arrays. In the tile-based environment, the control data consists of specific move commands, which are forward, backward, turn left and turn right. Based on these basic commands,

complex pattern movements are built. In other words, complex patrolling is a set of basic command blocks.

The simple and naïve method is defining each step on each command, for example, “turn forward 5 tiles, then turn right 20 degrees and continue forward 10 tiles.” It requires three blocks of commands to achieve this movement.

However, it is not flexible. If you want to walk from (34,67) to (102, 205), you do not know how many tiles need to go. To solve this problem, the Line-of-Sight algorithm is applied to it. Take Figure 2-9 as an example, a duck patrols in a square path.

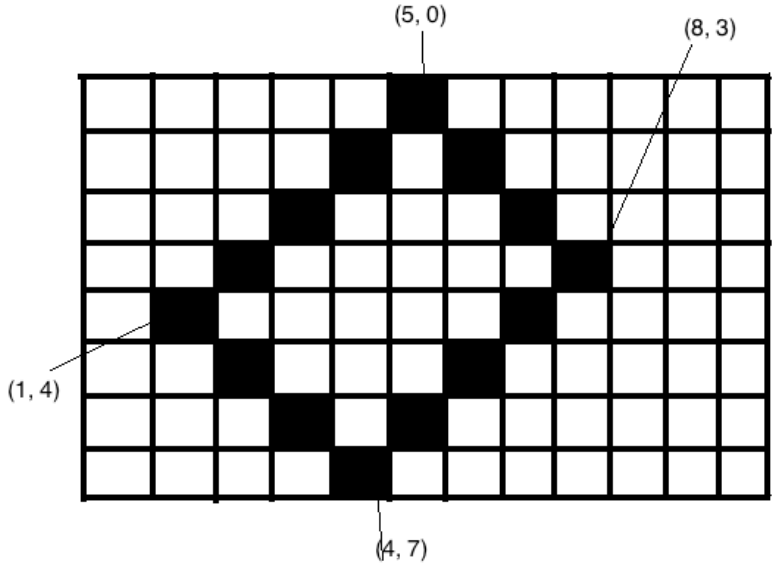

Figure 2-9: Duck pattern movement in square path

This is a top-left to bottom right coordinate system. The patrolling path is (1, 4) -> (2, 3) -> (3, 2) -> (4, 1) -> (5, 0) -> (6, 1) -> (7, 2) -> (8, 3) -> (7, 4) -> (6, 5) -> (5, 6) -> (4, 7) -> (3, 6) -> (2, 5) -> (1, 4). If using a basic pattern command array to define it, 4* 8 commands are required and it is harder to calculate the turning degree of each corner.

The Line-of-Sight algorithm makes this easier. In this example, calling Line-Of-Sight function four times will do it.

1. LineOfSight(1, 4, 5, 0);
2. LineOfSight(5, 0, 8, 3);
3. LineOfSight(8, 3, 4, 7);
4. LineOfSight(4, 7, 1, 4);

Input of line one: (1, 4) and (5, 0)
Output of line one: (1, 4) -> (2, 3) -> (3, 2) -> (4, 1) -> (5, 0)

Input of line two: (5, 0) and (8, 3)
Output of line two: (5, 0) -> (6, 1) -> (7, 2) -> (8, 3)

Input of line one: (8, 3) and (4, 7)
Output of line one: (8, 3) -> (7, 4) -> (6, 5) -> (5, 6) -> (4, 7)

Input of line one: (4, 7) and (1, 4)
Output of line one: (4, 7) -> (3, 6) -> (2, 5) -> (1, 4).

In the Jurassic Ducks project, the three functions, “initializePath”, “buildPath” and “normalize” in jdalgorithm.cpp implement the ducks’ pattern movement in a tile-based environment.

Pattern movement in a tile-based environment is similar to a physically simulated environment. The only difference is the commands. In the command data, “portSideThrustActive”, “starboardSideThrustActive” are used instead of “turnLeft” and “turnRight”. These two will activate the thrust on the duck to make the duck move left or right.

In the Jurassic Ducks project, this is combined with flocking behaviors. Therefore it is implemented in flocking function in jdalgorithm.cpp. Please see the next section where more details are given.

2.3 Flocking

The chasing and evading algorithm for each duck yields impressive results. However, that algorithm is used on each individual duck instead of a group. In this section, I will describe how the ducks move in flocks rather than independently. Further, I will show the obstacle avoidance technique based on the flocking algorithm.

2.3.1 Flocking rules

The classic flocking technique has three main rules (David and Glenn, 2004):

- Cohesion: each duck steers to the average position of its neighbors.
- Alignment: each duck steers to align itself to the average heading of its neighbors.
- Separation: each duck steers to avoid touching its neighbors. (This rule can be also applied in the obstacle avoidance algorithm.)

In the first rule, basically each duck is aware of its local surroundings. It should also know the average position and average heading of the flock. Figure 2-10 shows a duck' visibility.

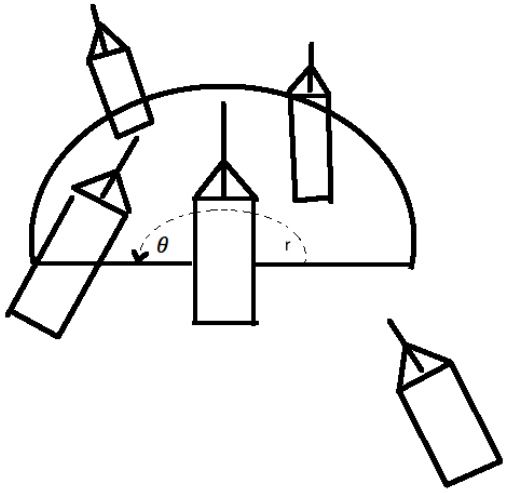

Figure 2-10: Duck visibility

In this figure the duck's visibility is illustrated as a semicircle with radius r. It is an option that the angle, whose default value is 180 degrees, can be changed in the Jurassic Ducks program. The widest field of view is 360 degrees. A wider field of view will make the flock much wilder, and a narrow field of view will lead to the formation into a line.

The main steps to implement flocking are below:

1. Find neighbors.
2. Calculate the average position and average heading direction, and average velocity.
3. Perform the flocking rules on each duck based on its visibility.

In the first step, local coordinates are used again. Figure 2-11 shows a duck's wide field of view in its local coordinates. It can be easily seen that if other ducks are in (x0, z0) and $z0 > 0$, it will be counted as this duck's neighbor.

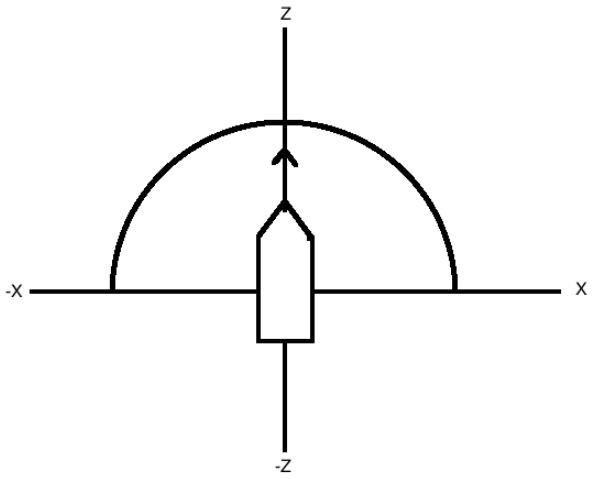

Figure 2-11: duck detects its neighbors in its local coordinate

Then the second step is simple. The average speed is a vector, which is equivalent to the total speed divided by the number of its neighbors. So is the average position.

The last step is performing the flocking rules on each duck. Taking cohesion as an example, the Figure 2-12 illustrates the average position of neighbors and the average heading of neighbors.

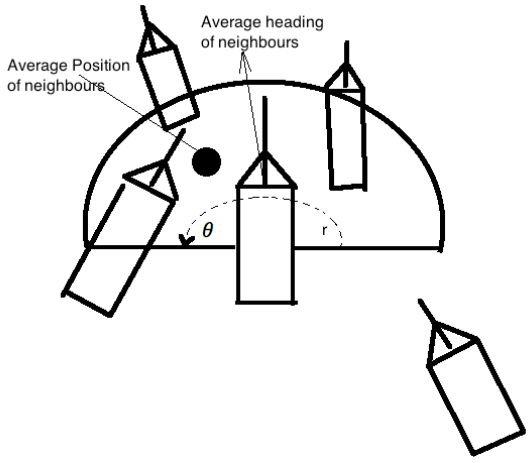

Figure 2-12: Average position and heading of neighbors

Using Line-of-Sight chasing is a good way to make the current duck move closer to the average position. Another way is to use Equation 2-5 (David and Glenn, 2004)

$$P_{thrust} += I_{steeringForceFactor} * \frac{\text{acos}(v * u)}{Pi} \qquad \text{Equation2-5}$$

P_{thrust} is the total steering thrust force on the ducks. $I_{steeringForceFactor}$ is a constant variable that controls the changing speed. □ is the duck's velocity vector and u is the average position minus the current duck's position. Therefore, $\text{acos}(v * u)$ is the offset angle between the duck and the average position. In this way, the further the duck is from the average position, the larger the thrust steering force added to the duck.

In the Jurassic Ducks project, $I_{steeringForceFactor}$ can be changed when the program running. If the players feel unhappy with the default $I_{steeringForceFactor}$ value, say they think the ducks' direction changing speed is too fast; they can customize it by using a setting box (Figure 2-13). The third value is for setting steering force factor.

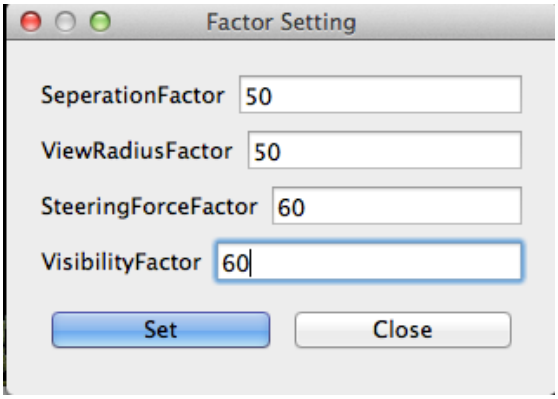

Figure 2-13: Flocking factors setting box

The second value in this figure is for setting the radius of the duck's view. The other two values are used for setting the separation threshold and the visibility distance for the duck (used in obstacle avoidance). The player can also customize all of these.

The cohesion rule make the closer ducks move as a flock. However, it will make the ducks move closer and closer to each other. Therefore, a separation command is required. Separation implies that the computer-controlled ducks need to separate if they are moving much closer to each other. It is simple and similar to the cohesion

algorithm. When the distance between the duck and its neighbors is less than the separation factor Equation 2-6 (David and Glenn, 2004) is used.

$$P_{thrust} += I_{steeringForceFactor} * \frac{I_{seperationFactor}}{D} \quad \text{Equation 2-6}$$

Here, the D is the distance between the duck and its neighbor. $I_{seperationFactor}$ can also be set by the user in the Jurassic Ducks program as Figure 2-13 shows.

2.3.2 Obstacle avoidance

In the Jurassic Ducks project, some trees are set as obstacles on the map (Figure 2-14).

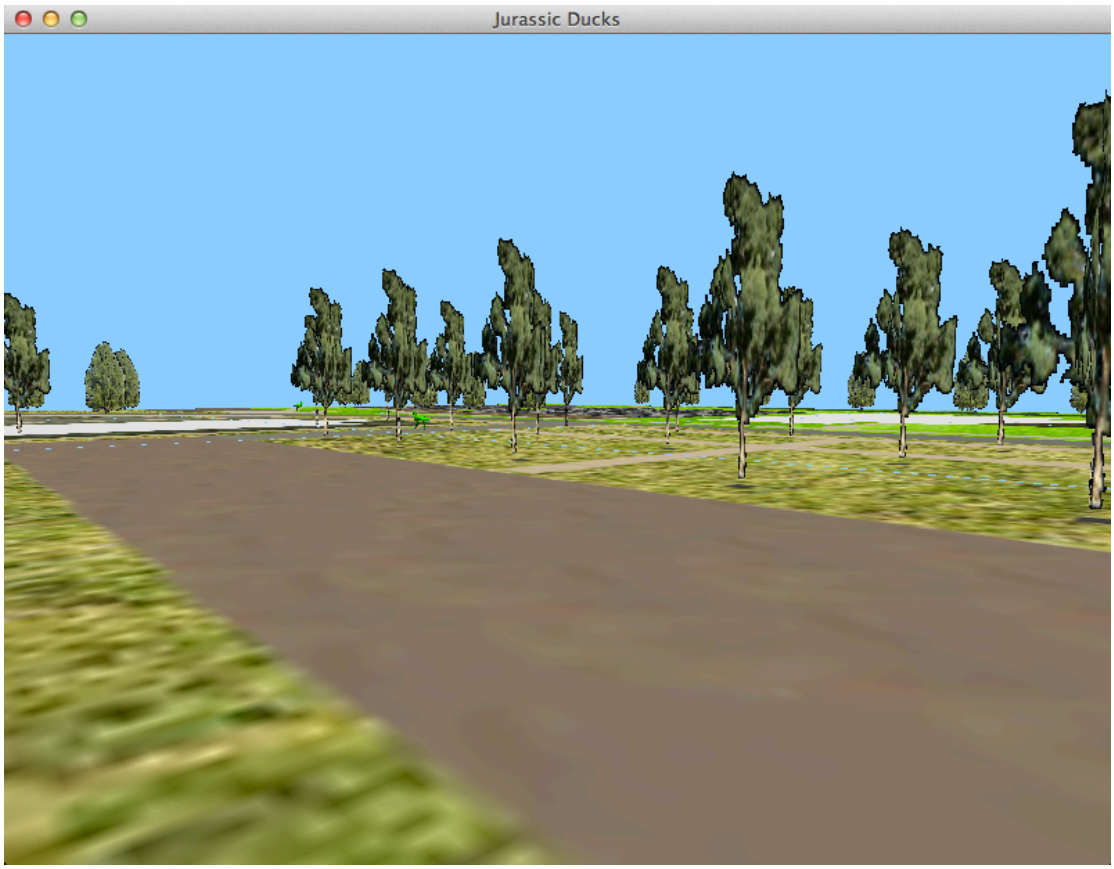

Figure 2-14: Trees as obstacles on the map

The ducks need to avoid these trees while moving, and this is easy to implement with flocking rules. Take a look at Figure 2-15 to see how obstacle avoidance works in the Jurassic Ducks project.

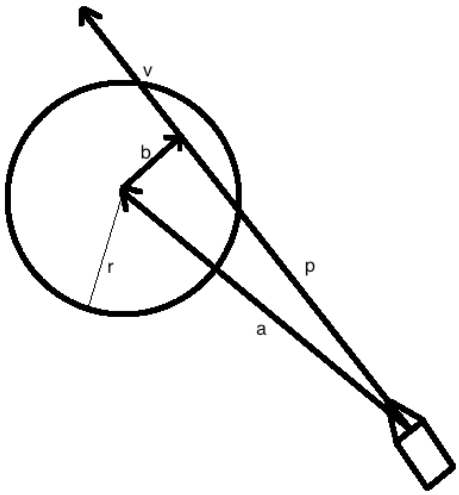

Figure 2-15: Obstacle avoidance

The circle is the obstacle with radius r. V represents the duck's visibility distance and direction. b is a vector point from the duck to the obstacle center, which represents the difference between the duck and the obstacle's position. Then projecting a onto v by taking the dot product yields p. b is calculated from the subtraction of p and a. Now there are two conditions to test whether the duck will intersect the obstacle or not. One is that the magnitude of p must be less than the magnitude of v, in other words, the obstacle must be in the duck's visibility area. The other one is that the magnitude of b must be less than the obstacle radius. If both of these conditions are met, the duck will hit the obstacle. Therefore, we can change the duck's steering force before it hits the obstacle. Equation 2-7 (David and Glenn, 2004) is used to add this steering force.

$$P_{thrust} += I_{steeringForceFactor} * \frac{I_{visibilityFactor}}{A} \qquad \text{Equation 2-7}$$

The $I_{visibilityFactor}$ is the visibility distance of the duck. A is the difference in distance between the duck and the obstacle.

The constraint of this obstacle avoidance algorithm is that the obstacles must be cylinders. In other words, the obstacles must have radius and circle center.

In the Jurassic Ducks program, the function flocking in jdalgorithm.cpp implements the three rules of flocking behavior.

2.4 Abandoned AI algorithms

Besides Chasing & Evading, Pattern Movement and Flocking AI algorithms, other AI algorithms can also be applied in games. For example, David and Glenn (2004) also mentioned Fuzzy Logic and Genetic Algorithms and Neural Networks in their book. However, those are not used in the Jurassic Ducks project. The reasons are below.

Fuzzy Logic (David and Glenn, 2004) deals with reasoning that is approximate rather than fixed and exact. In multiplayer online games, fuzzy logic can be implemented to rank the players' and non-players' characters in terms of their combat prowess. The player can base this rank on factors such as weapon, armor, strength, agility, magic power etc, and then combine them together to yield a ranking like easy, moderate, or tough. The Jurassic Ducks project is not an online game. Therefore, the fuzzy logic algorithm was abandoned. However, it could be helpful for changing this game into an online game.

Genetic Algorithms (David and Glenn, 2004) are search heuristics that mimic the natural evolution process. It is often implemented in multiplayer role-playing games. In those games, the player is able to choose different types of character classes. When they kill an enemy, the enemy may drop some weapons. However, this weapon type may not be suitable for the current characters but it could be useful later. For example, a warrior may get a magic wand. A genetic algorithm deals with this problem. As the Jurassic Ducks project is not a multi-role game, this algorithm was not used in the game.

Neural Networks (David and Glenn, 2004) mimic the biological neural networks that can automatically generate the result from the input after training enough times. In the Jurassic Ducks project, obstacle avoidance has a constraint that it can only deal with cylinder obstacles. In other words, if there are some square walls, the algorithm does not work. In a future version, neural networks could solve this problem. The basic idea would be building an neural network, then training the ducks in the map with some walls and adjusting the duck's moving behaviors based on the rule (the duck shall not touch the walls).

3. Computer Graphics

This section will describe what OpenGL features were used in the project and what features were tried but not used. It will also show the problems I encountered during the development period.

3.1 AI Testing Environment with OpenGL Rendering

In the first seven weeks, I implemented the AI algorithms in the Jurassic Ducks project. There are two ways of testing these algorithms. One is creating some console applications to test them. The Line-of-sight chasing and pattern movement algorithms' testing console applications are in the *Test* Folder. However, some algorithms were hard to test in the console. For example, the player's real-time movement is hard to implement in a console application. The other way of AI testing is using the game itself.

To simplify the testing environment, the ducks and player were drawn as green cubes. The cube's vertex data and surface normals were constructed in arrays. I use the glBegin and glEnd functions (Richard et al., 2007), which delimited the vertices of a primitive or a group of like primitives. (Please see jduck.cpp file). The obstacles were implemented as red spheres (please see jdobstacle.cpp). The terrain was only implemented as a rectangle (please see jdterrain.cpp).

The player's view was implemented with a Camera class (please see jdcamera.cpp). Actually, there are two types of view, one is the first-person view and the other is the bird's-eye view. Both of them are objects of the JDCamera class. The difference is that the first-person view object is initialized in the player's standing position and looking to the front; the bird's eye view object is initialized in a higher height position overlooking the terrain (please see the JDWidget class's constructor in jduckswidget.cpp). The user can change to the bird's eye view by holding the "v" button in the keyboard (please see keyPressEvent function in jduckswindow.cpp). The user could also change their point of view in pitch and yaw by dragging their mouse on the game application window. The program will detect the moved distance and direction and change the view based on them (please see mousemove function in jdcamera.cpp).

OpenGL can simulate different types of lighting (Richard et al., 2007) such as point light, directional light etc. For Jurassic Ducks project I chose the directional light (please see jdlight.cpp) instead of a point light. The main difference between these two is that a directional light does not have a position, so the rays of a directional light

are all parallel, while the rays of a point light radiate outwards from that point. The reason for this choice is that the sun does act more as a directional light than a point light with respect to the earth, because it's so far away.

The standard OpenGL rendering pipeline is created in jduckswidget.cpp. The *initializeGL* function creates the 3D environment and loads the map. The *paintGL* function is called whenever the widget needs to be painted (draws the world).

3.2 The map loader

After I finished AI testing, I implemented the real map instead of a test rectangle in the Jurassic Ducks program. My supervisor Mr. Hugh Fisher created the map texture using Google Earth. The texture map (CSIT/mapTexture.png) is a 1024*1024 PNG picture, which applied to the entire map. The type image (CSIT/mapType.png) bounds are 204.8*204.8. In other words, each pixel represents 0.2 meters in the real world. Further, each pixel in the type image represents a given terrain type: building, road, grass etc.

When creating the map, the map is broken into smaller chunks (please see createTiles function in jdmap.cpp). This can make rendering more efficient. It uses the OpenGL vertex, color and texture coordinate arrays (Richard et al., 2007) to save map data (please see jdmap.cpp). This is because the map data is much larger than the single cube. The glDrawElement function (Richard et al., 2007) is used instead of glDrawArrays (please see render function in jdtile.cpp). This is because the size of data sent to the graphic card is different.

DrawArray :
sizeof(vertex) * the number of triangles * 3

DrawElements :
sizeof(vertex) * the number of vertices + sizeof(indice) * the number of triangles * 3

Therefore, in many cases, the draw element is much faster.

The map is rendered either as wireframe (Figure 3-1) or as lit & shaded (Figure 3-2). The program supports menu items to change these two rendering styles when debugging.

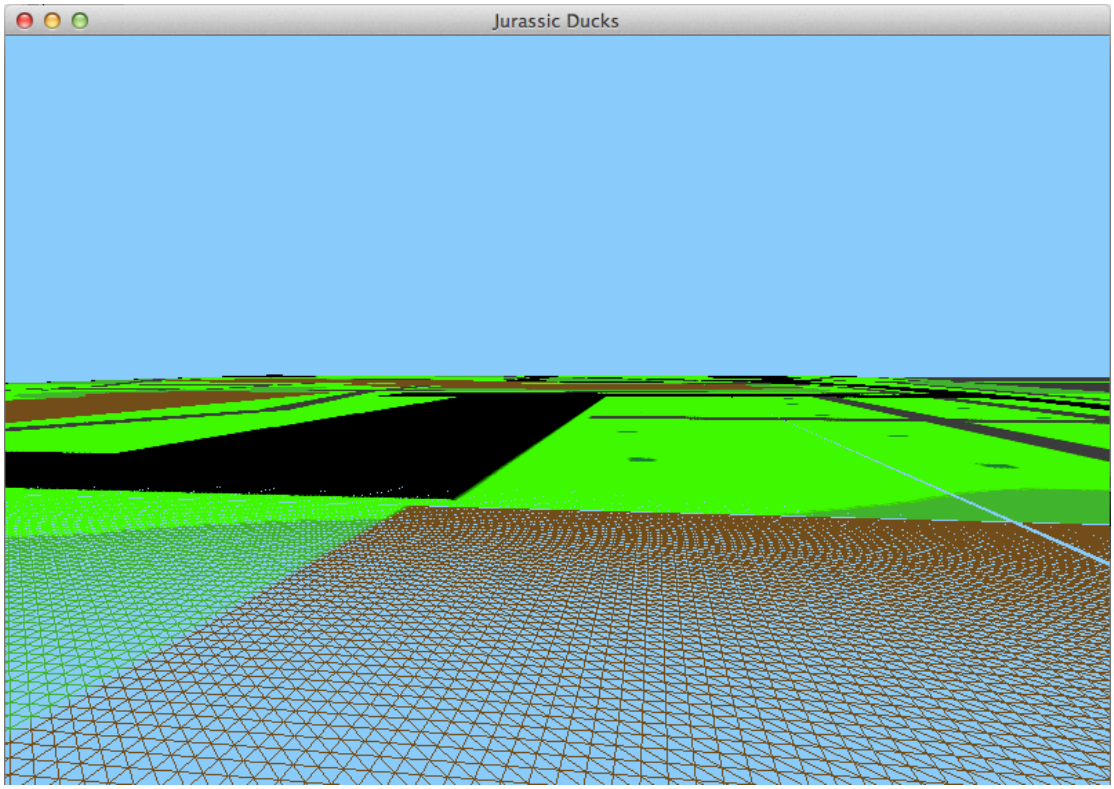

Figure 3-1: The wireframe map

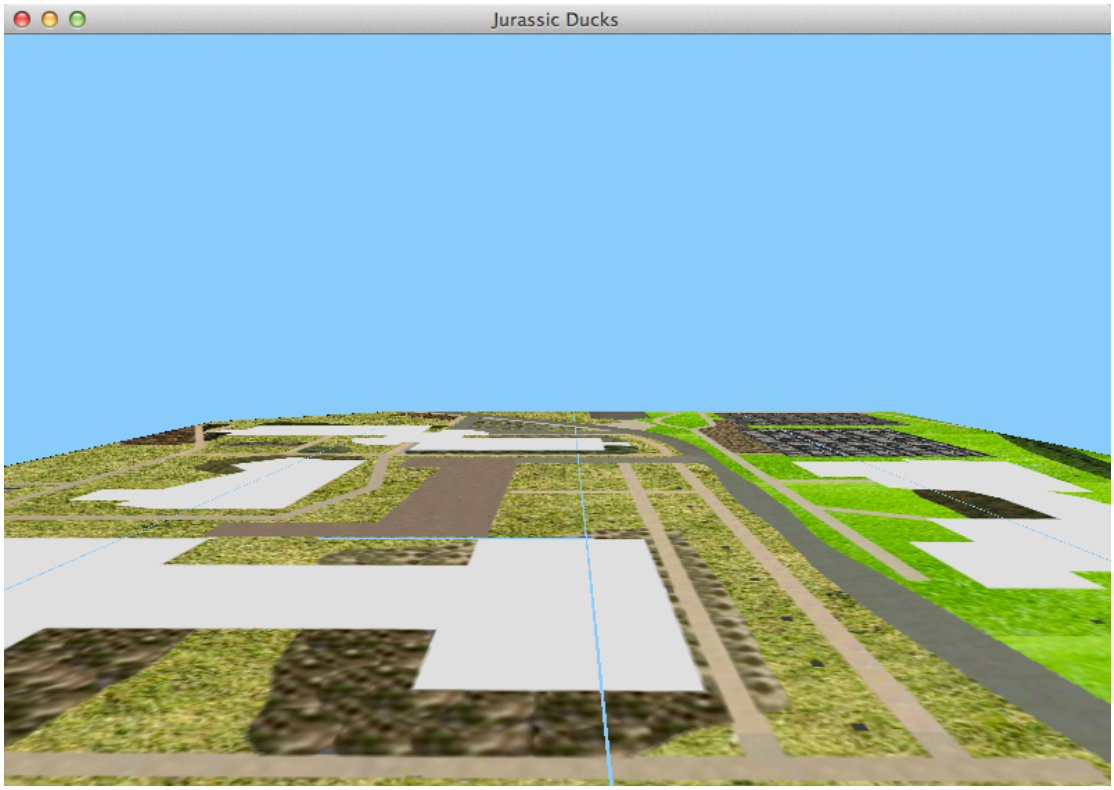

Figure 3-2: The lit & shaded map

There is a problem in Figure 3-1 and Figure 3-2 that is there are cyan strips between the tiles of terrain. These are almost certainly caused by an off-by-one error in the tile creation code, the last row and column of triangles on each tile needs to be extended to meet the edge of the next one across / down. The Figure 3-3 shows the correct map.

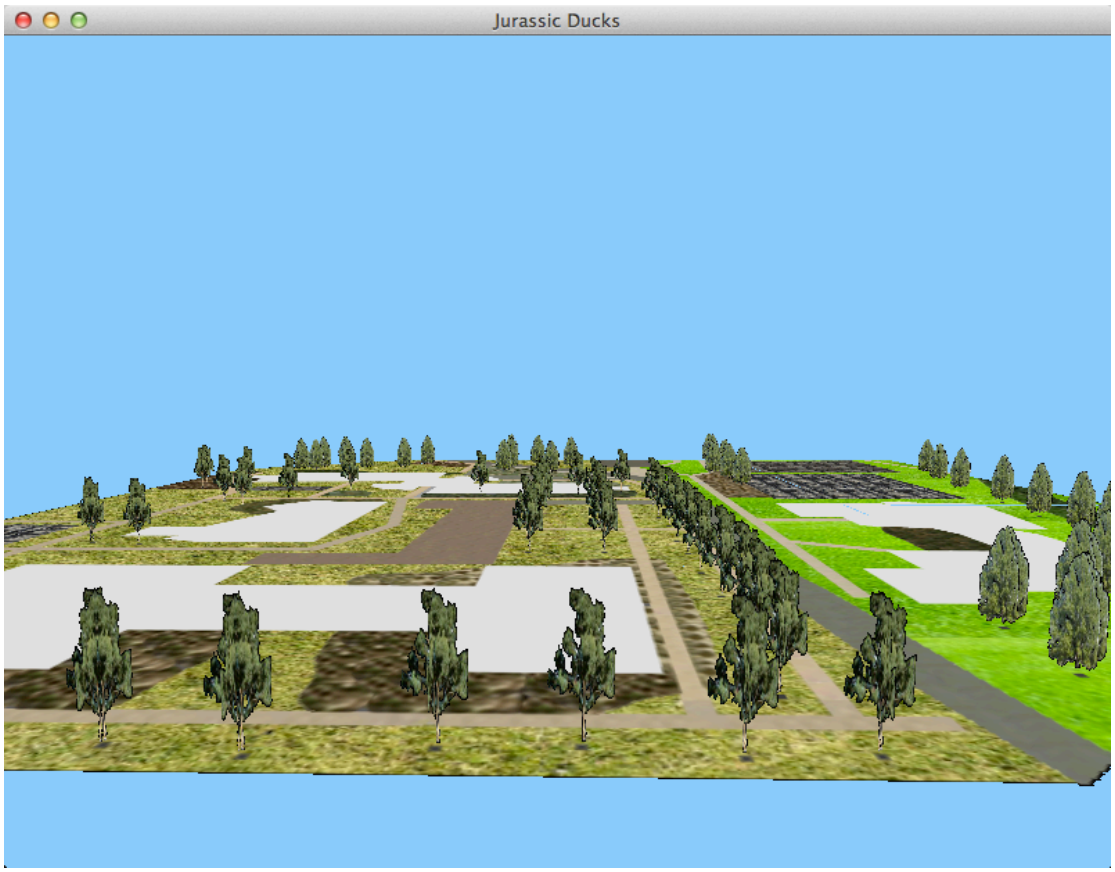

Figure 3-3: The map without cyan strips

There is also a tricky thing when loading an image with Qt Image (Jasmin and Mark, 2008). The QImage byte order is BGRA (blue, green red and alpha channel). The byte order is different to usual for OpenGL so when using glTexImage2D the byte order must be GL_BGRA (please see loadTextureMap function in jdmap.cpp)

3.3 The Duck Model and Tree Models

After finishing AI testing, I replaced the cube (the duck) with a real duck model with surface normals for smooth shading (please see jduck.cpp). The last few lines of the update function in jduck.cpp show how to make the duck's legs swing back and forth, a simple walk cycle. The walk cycle is controlled by the speed of the duck rather than the frame loop. Figure 3-4 shows the duck model.

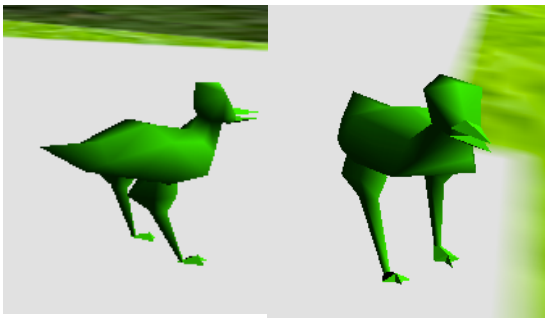

Figure3-4: The duck model

For efficient rendering, the trees are displayed as a pair of quads with a texture map (please see jdtree.cpp). A JDTrees class (please see jdtrees.cpp) is a batch of trees sharing the same texture. Figure 3-5 shows the tree texture.

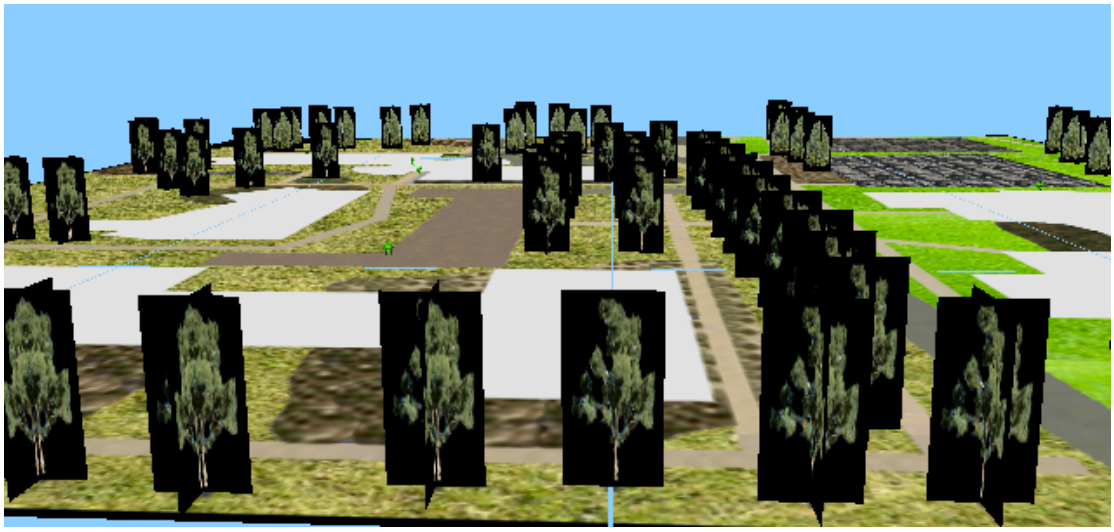

Figure3-5: The tree model on the map

It looks strange that all of the trees have a black background. This is because the QImage (Jasmin and Mark, 2008) loader doesn't handle transparency. The solutions I chose is writing a fragment shader script (Richard et al., 2007) to hack it. The next section *GPU calculation with GLSL* will show this solution.

3.4 GPU Calculation with GLSL

The standard OpenGL rendering pipeline yields impressive results. However, nowadays most computers have a GPU with new feature added to allow for increased flexibility in the rendering pipeline at vertex and fragment level. Programmability at this level is achieved with the use of shaders. In the Jurassic Ducks program GLSL 1.2 (Richard et al., 2007) is used to implement these two shaders.

In section3.3, there is a problem with the tree texture that all of the trees have black backgrounds. A fragment shader script will solve this problem (please see std_frag.glsl). In that script, the fragment shader will check if the alpha channel is equal to 0 or not. If it is equal to 0, the graphic card will not render the texture pixel. However, the QImage loader doesn't handle transparency, so the fragment shader script checks if the RGB channels are equal to black color, and if so discard it. Figure 3-6 shows the correct tree texture.

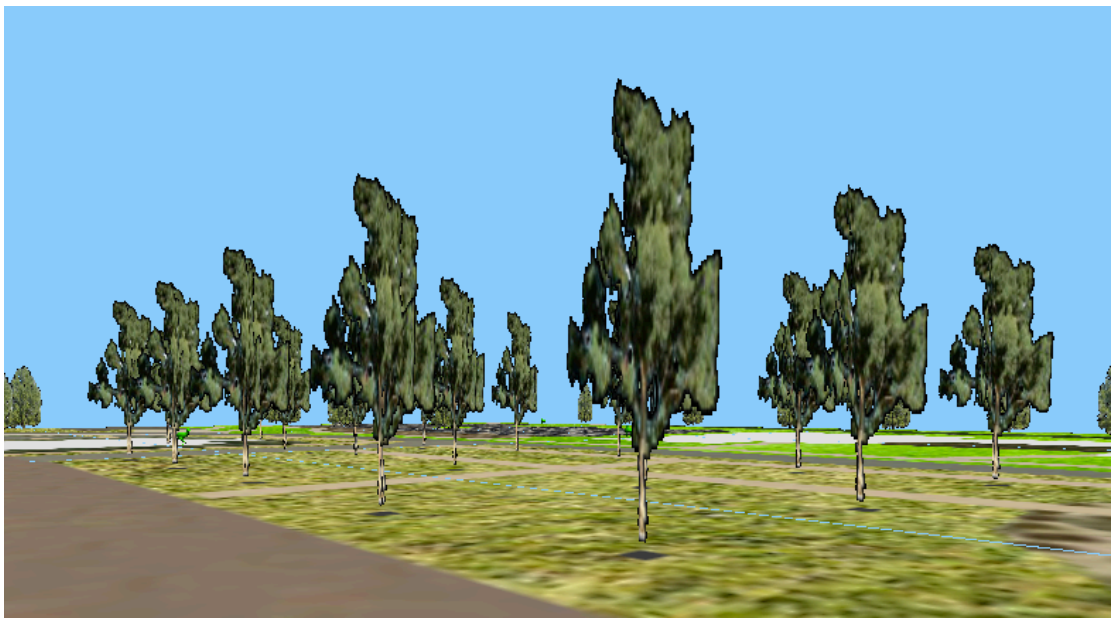

Figure 3-6: The tree model on the map without black background

The GPU shader loader is implemented in jdgpushader.cpp. Figure 3-7 (Richard et al., 2007) illustrates the steps required to create GLSL shader objects and link them together to create an executable shader program.

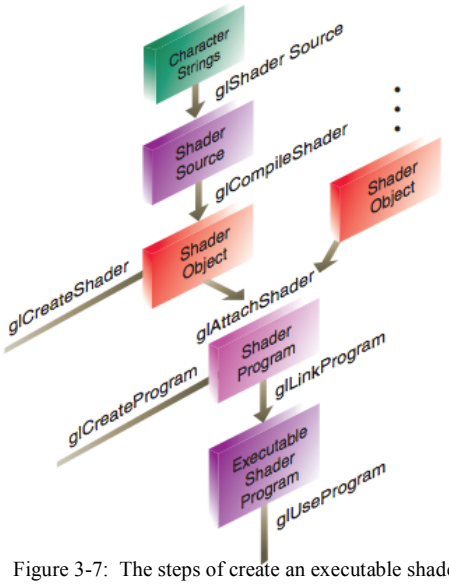

Figure 3-7: The steps of create an executable shader program

There is an error message in the Jurassic Ducks program when calling this shader loader, that the current draw framebuffer is invalid. The game calls the shader program before initializing OpenGL may be the cause of this. However, it seems not to be a problem in the Jurassic Ducks program.

To sum up, not only are modern GPUs very efficient at manipulating computer graphics, but also they can make some 3D rendering effects that the fixed rendering pipeline cannot do.

4. Software Framework

In this section, I show the game program's main design structure. If you want to know more detail about the class diagram of the Jurassic Ducks project, you can go to appendix 1.

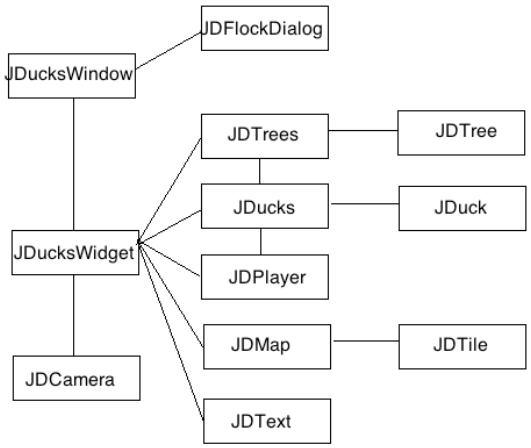

Figure 4-1: The program structure

Figure 4-1 illustrates the program's structure. The game starts from JDucksWindow's initialization. The JDucksWindow consists of a canvas (JDucksWidget), which displays the game on the screen and a menu bar with some menu items. The menu bar class and menu items class are not shown on Figure 4-1, because they are QT built-in classes. One of these menu items pops up a dialog box (JDFlockDialog) that is used to help users to set the flocking factors. The canvas contains a set of ducks (JDucks) and trees (JDTrees), a player (JDPlayer), a map (JDMap) and a text message display module (JDText). The map is broken into smaller chunks (JDTile).

5. Conclusion

In conclusion I got a deeper understanding of game AI algorithms, computer graphics and game development in software engineering. Especially I gained a lot of AI algorithms design, because many general computer graphics books cover little of game AI algorithms and neither do the AI books. People learning both Computer Graphics and AI at the same time, and applying them together is a good way of studying. Further, AI in game development is more and more popular in the human computer interaction area. While developing the Jurassic Ducks program, I learnt to fix some tricky errors and bugs. Other game developers may encounter these errors, so this report could be helpful for them as well.

In future work, the project could be extended by others in many ways. First, they could add more buildings and obstacles into the map. Second, they could extend both the player behaviors (shoot, defend, pick up object) and the duck's behaviors (swim in river, walk on ground). Third, they could improve the shader programs so that the game appears more photorealistic. Fourth, they could change the game into an online game. Many advanced AI algorithms would be really helpful (please see section 2.4 Abandoned AI algorithms). Further, the course COMP8420/COMP4660 Bio-inspired Computing: Applications and Interfaces from Australian National University taught by Dr Tom Gedeon would be really helpful for this project.

Appendix 1

The Jurassic Ducks program class diagram.

Reference:

Andreas Pfeifer, “Creating Adaptive Game AI in a Real Time Continuous Environment using Neural Networks” Technische Universität Darmstadt, Department of Computer Science Knowledge Engineering Group, 2009

Bresenham, J. E., "Algorithm for computer control of a digital plotter". IBM Systems Journal 4 (1): 25–30, 1 January, 1965

David M. Bourg & Glenn Seemann, “AI for Game Developers”, O’Reilly Media Inc, pp6-79, pp188-348, 2004

Dave Shreiner, “OpenGL® Programming Guide Seventh Edition The Official Guide to Learning OpenGL®, Versions 3.0 and 3.1”, Addison-Wesley, 2009

Jasmin Blanchette and Mark Summerfield, “C++ GUI Programming with Qt 4, 2nd Edition”, Prentice Hall, 2008

Richard S. Wright, Jr. Benjamin Lipchak Nicholas Haemel, “OpenGL® SUPERBIBLE Fourth Edition, Comprehensive Tutorial and Reference”, Addison-Wesley, 2007