Department of Electronic Engineering

FINAL YEAR PROJECT REPORT

BEngCE-2006/07-SYY-31-BECE

AI Role Playing Game Development

Student Name: Chow Ying Kit Ricky
Student ID:
Supervisor: Dr. S.Y. Yuen
Assessor: Dr. H.C. So

Bachelor of Engineering (Honours) in
Computer Engineering

Student Final Year Project Declaration

I have read the student handbook and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I declare that the work submitted for the final year project does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Student Handbook.

Project Title: AI Role Playing Game Development

Student Name: Chow Ying Kit Student ID:

Signature Date:

Acknowledgements

I would like to thank all the people for their generosity who have given me useful supports in my project development.

First of all, I greatly appreciate the kindness of my supervisor, **Dr. S. Y .Yuen, Kelvin** who gave me an opportunity to work for my interest. His great patience and sincere discussion helps me to achieve my goal in this project. Without his professionalism and enthusiastic support, my project is nothing and I would not have a chance to learn and implement AI in this game development.

Hereby I would like to express my thanks to a research assistant, **Mr. C. K. Chow** for his substantial suggestion and support. Moreover, I would also like to thank my assessors, **Dr. H.C. So**, my fellow schoolmates, my friends, family and people from the internet for their kind assistance and encouragement.

I would also appreciate that I could take the IAS training in **Hamster Force Multimedia Limited Company**. From there, I have learnt some skills of commercial game programming and I acquire some graphics design technique in Photoshop and 3DSMax for the sake of my project.

Through this FYP, I have learnt a lot of techniques in making game which is a good experience for me and it is useful for my future career as game developers. I hope this project can be a helpful handout or references for the people like me who eager to develop game in order to motivate the game industry in Hong Kong.

Table of Contents

AI Role Playing Game Development

Name : Chow Ying Kit Std.Id :

1. Abstract

In spite of recent great advances in computer gaming industry, the artificial intelligence (AI) ability of present day games is still weak and not desirable. AI usually consists of cheating by the computer or is hard coded with no learning ability. Developing good AI techniques that is intelligent, entertaining to the players, and with good learning ability is a subject of great interest in both the academic community and the industry.

This project develops a three dimensional role-playing (RPG) action game about a player fighting a small group of monsters using recent computer graphics techniques such as bill boarding and ray picking. A neural network (NN) approach is developed to train the monsters in the game. The behaviour of the player is used to train the monsters using online learning of the weights of the network.

The developed game demonstrates the ability to learn interesting manoeuvres used by human players to attack or defend against the monsters.

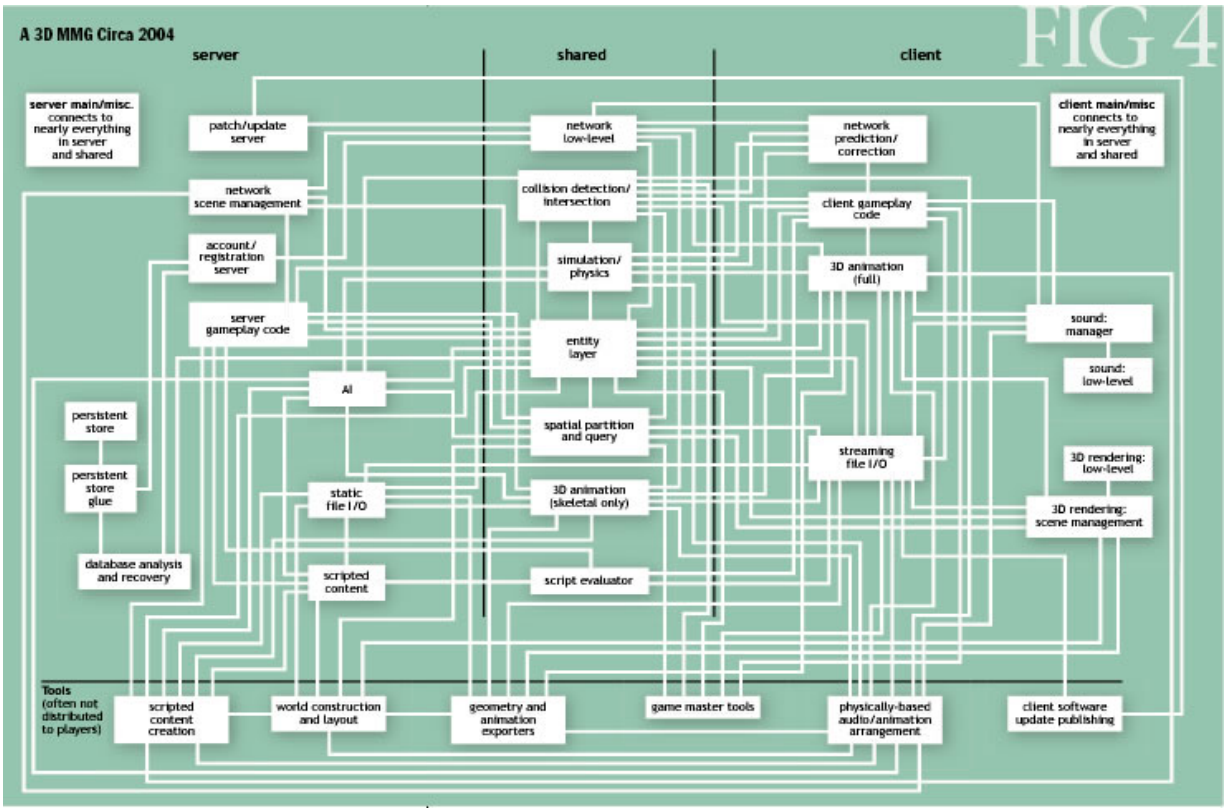


Figure 1a) The complexity of the game components in an RPG nowadays

2. Objective

The objective of this project is to develop a Role-playing action game engine which illustrates the features of game architecture, computer graphics, and artificial intelligence.
There are 3 main criteria in this project:

- RPG system design

  Sophisticated portal system is required in the project. Characters are assigned to have the basic ability of physical attacks and spells which are the common characteristics appeared in RPG.

- 2D-3D graphics representation

  2D in 3D graphics is a feature in this engine. In addition, the technique used in the game industry is greatly demonstrated.

- Machine Learning and evaluation

  Neural Network is applied in the project to demonstrate the behaviour of NPC in the game. The purpose is to let the machine learn and evaluate the action taken by players, followed by returning desirable counter-attacks and actions with heuristic thinking.

In results, a completed game demo is shown which is tailored made to show interesting manoeuvres by monsters increase its attractiveness.

3. Introduction

Role-playing game (RPG) is a type of game in which players assume the roles of characters and collaboratively create narratives.[1]. This sort of games not only requires computer graphics rendering, human-computer interaction, artificial intelligence, but mainly focus on file handling (map, script, scenarios or so forth) and characters data structures management.

This project is specified for the feasibility studies and evaluation of the techniques used in this genre of game with sub-divided into two main sessions: Graphics representation and Artificial Intelligence, where the two topics are illustrated with detailed implementation followed by evaluation and discussion.

4. RPG system design

Action Role-playing game system is applied in this project. This subgenre tends to be faster-paced, more skill intensive and focused on combat. Players assume the role of a fictional character (most commonly in a fantasy setting) and take control over many of that character's actions.(warrior/magician/ priest). The portal system requires the management of the player and monster characters, sets the rules and restrictions in battle, and monitoring of the monster characters to trigger AI, processes scripts and dialogues. To simplify the implementation, the project demo mainly focuses on the basic management of characters and executes fundamental character's actions. It includes a controller to control characters for dynamic purpose. A simple battle system whereas players fight with monsters with basic hit and spells. HP (Health Points), MP (Mana Points), and time are counted as an RPG issues for AI decision making or players' strategy planning.

4.1 Architecture of game

Normally computer game program is a kind of applications which uses the features of Window components for the driver of the program. Windows enable event-driven programming which saves a lot of problem without manually keep tracking of the controls and input interrupts, therefore it provides an efficient platform for any applications run on the OS. Unlike other application program, game platform tempts to apply traditional Win32 components instead of MFC components because of tempting to run the game program in a high speed and great performance. For graphics implementation, games normally apply either DirectX or OpenGL for the development, no matter the graphics genre is 2D or 3D, in order to adopt the standard plug-in of graphics adapter and windows environment.

Better Game architecture support better game management and maintenance. (Figure 4) A definition file, commonly called"game data" is the component of how the way game behaves and it is the database to manage resources, control game status, and trigger all other components to work. With the supplementary of Physics Engine, Logic Engine, Event Handler and other configuration parts linked to this game data core, the entire system can then handles the game play, simulates the world and responds to user's input. The request is then passed to the hardware abstraction layer for input, graphics and audio handling and finally processed by hardware.

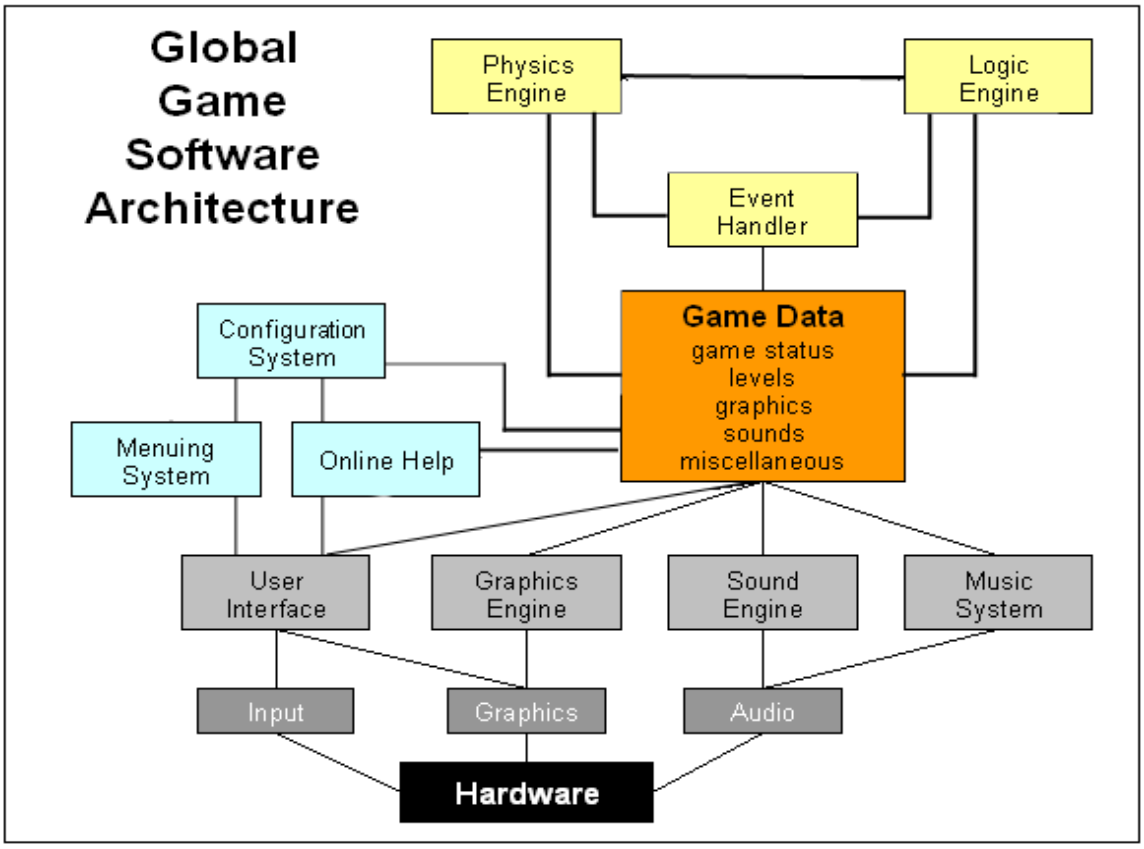


Figure 4) Global Game Software Architecture

To program a game in Microsoft Window, the first step is to familiarize with basic Window programming and set up a window for display. The origin of a Window program is the **WinMain()** function where the attributes of the game window is prepared. Since windows is event-driven, the program falls into an iterative "while loop" to pump windows message for execution and the loop is the core of game application. Developers define game functions which are trigged and run within this game loop.

For better programming structure management, the Win32 functions are encapsulated in a game system class and all the game components are programmed using the principle of OOP. The

idea behind is to create the game objects with more robust and clear code with the benefit of high reusability. In the demo program, about 50 cpp and 50 header files are split in this game engine. Among them, **Game.cpp** and **Game.h** file are the main paired which implement the main entry of the game logic. The file holds a bulk of the game's logic which trigger other file functions and this is invoked by **GameSys.GameMain()**. Apart from the accessor functions in **cGameApp, GameMain()** is publicly accessed so that every components can share the resources in the core and this entire system is **Game**.

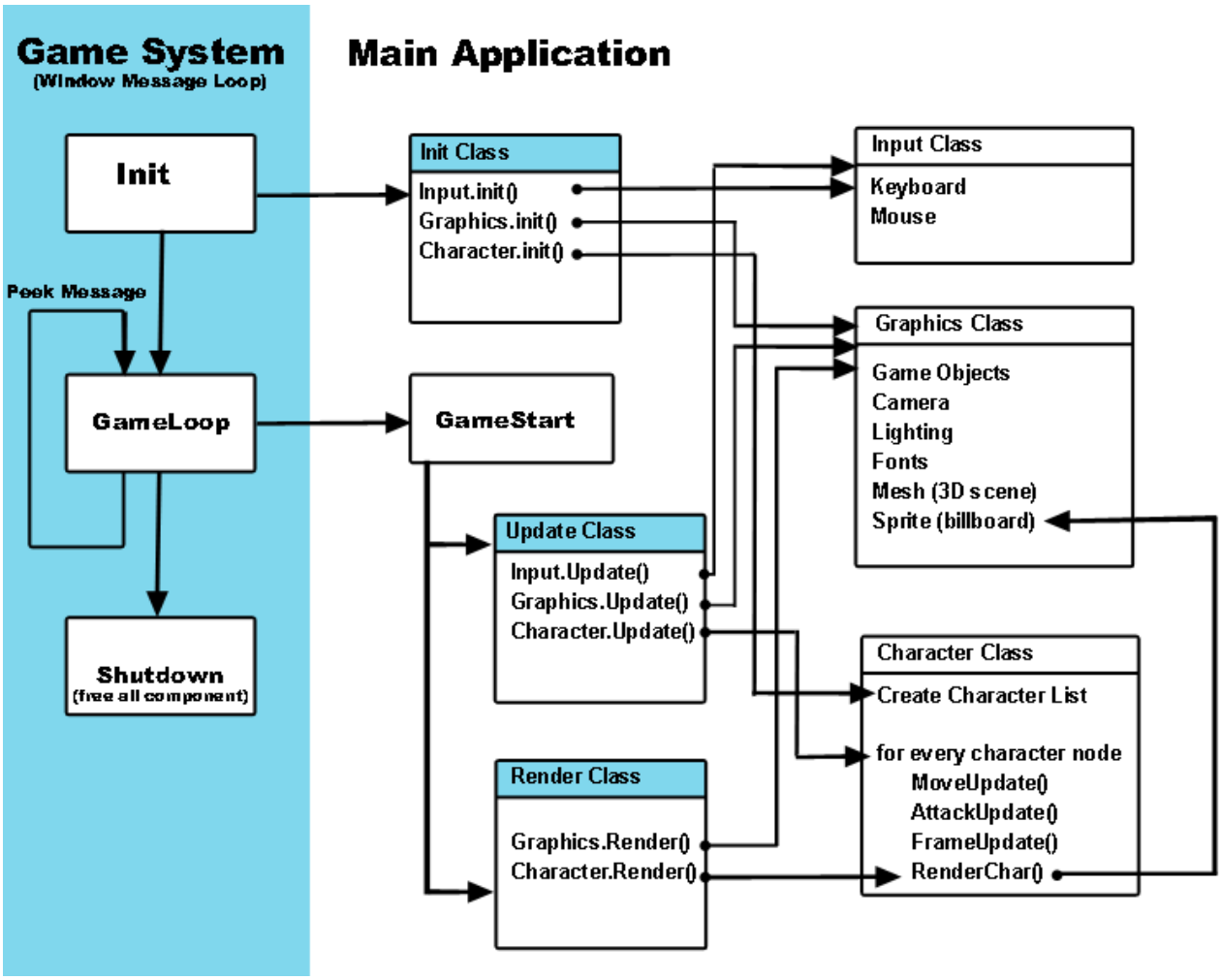


Figure4.1) The Basic Framework and structure of game system

In game program structure, there are 3 main components which form the framework : **Init, Update, Render**.(Figure 4.1) Init class processes the initialization of graphics, input, audio, game components (AI, players, monster) while Update and Render edit the parameters used in game and draw the graphics respectively. The Update and Render class continues in a loopy process which the components in the class are called every elapsed time. Since game is advancing into larger size and complexity, there is no reason to use one class for serving all the functions because of inefficient management and hard to debug. In case, the classes are defined and separated for its main operation, defined into components (Init, Update, Render) which are called every time by other processes. For instance, Character class defined its own function (Init, Update, Render) and process independently which it is called by other classes.

4.2 Design of Character Structure

To extend the program for later development and maintenance, good management of game parameters (players, monsters, effect, attack, attributes) is required. Normally a class is used to handle all the actions and attributes of a character structure and generate the character into a list. The class, so called “Character Controller”, monitors and controls all characters (no matter players, monsters, NPC, remote player) into same set of data structure. Typically, a linked list is used to store the sets of characters for easy insertion and deletion (Figure 4.2) whilst some research states that tree stores the characters in a faster way and higher performance.

Character Structure should contain important information of each character ID, character Type, coordinates, direction which is the main component for system to identify and manipulate the specified character. Attributes in Character structures is arbitrary designed and it is determined by the gameplay and game genres. In general, a basic ARPG demonstrates the combat between players and monsters to calculate the damage of characters or reward for characters, that is, health points and maximum health points is crucial in this case.

The implementation of Character List also encoded with the three main functions in the program structure: **Initialize, Update, Render**. During initialization, character list is generated according to the number of character input defined by developers and every character is declared a unique character ID for further recalls. While updating, the list is run iteratively which is prompted by character controller. Movements, actions and attributes of characters are updated within an elapsed time. Finally, the list is surfed through to render the desired character on the screen.

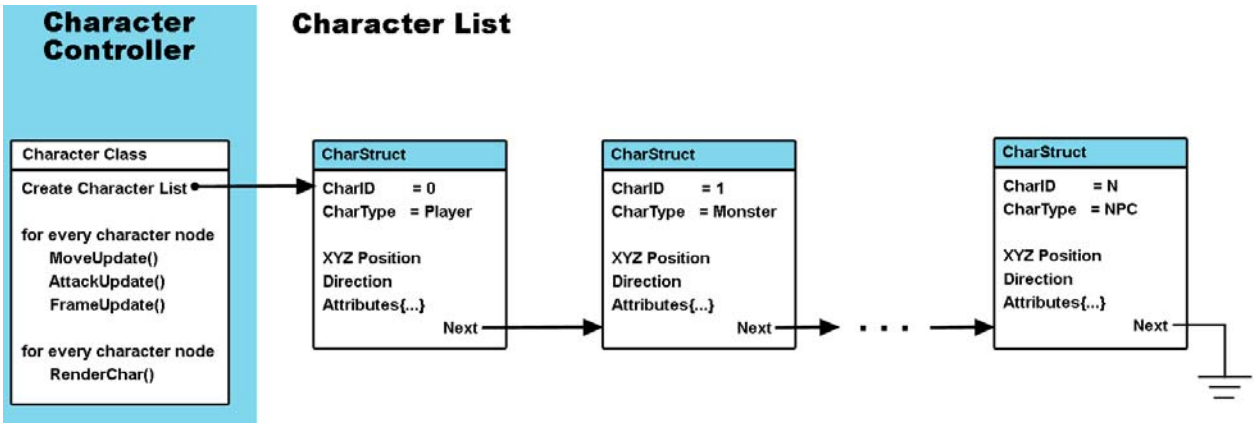


Figure4.2) Basic Structure of Character Controller. Notice that the all characters are linked in one list with different character type and attributes.

5. 2D 3D graphics representation

2D graphics can be represented as 3D in game development. This section focus on the main features that commonly used in the commercial games – **Billboarding and Sprites (Section 5.1), Mouse Picking (Section 5.2)** which are used to render 2D graphics as a flat plane to map on a 3D screen and illustrates the used of mouse click for character manipulation respectively. The technique is also applied in the project demo with tailored-made algorithm for the rendering of character sprite.

5.1 Billboarding and Sprite

In definition, **Billboarding (Figure 4.1a)** is a rendering method which adjusts an object's orientation so that it faces the eyepoint (camera). The objects (so-called **sprites**/**imposters**) are only viewed from the same angle by always perpendicular to the axis emanating from the camera. The image can be scaled to simulate perspective, it can be rotated two dimensionally, overlap other objects and be occluded.

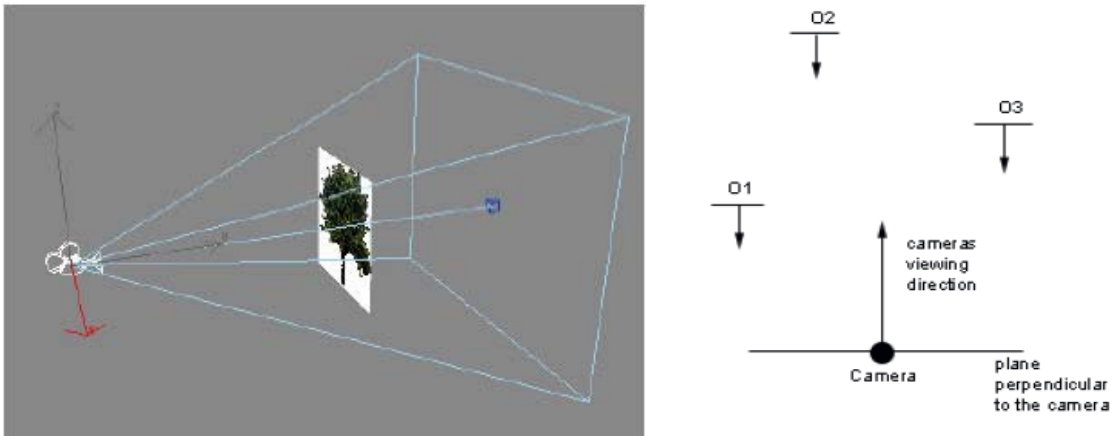


Figure 5.1a) Concepts of billboarding (the sprites are always facing the camera)

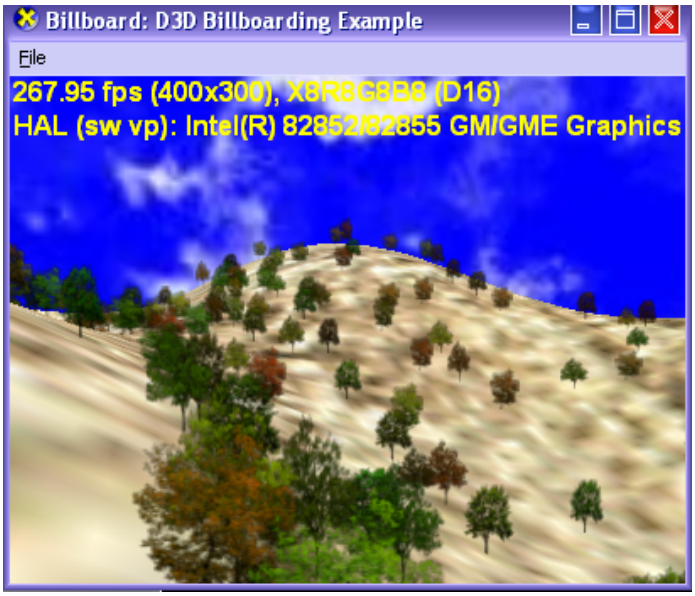


Figure 5.1b) Billboard tree example (resources from Microsoft DirectX 9.0 SDK)

Commercial games implements complicated graphics wisely by using billboarding. Clouds, explosion effects, magic spells, lens flare, snow drops use this technique frequently instead of traditional 3D geometric model or particle systems. This is due to prevent hurting performance and tempts to reduce the number of polygons. For instance, a decent representation of trees can be replaced with imposter texture sprites (2 triangles). **(Figure 5.1b)** The technique guarantees that the texture is always facing the camera, therefore the user never realizes that the effect or objects is in fact a flat texture quad.

Figure 5.1c) An example of sprite animation from the popular game - *The Legend of Zelda: The Wind Waker*. In this frame the sprouts of grass and brown puffs of smoke are integrated into the scene using sprites. Notice that one sprite at the bottom of the largest puff of smoke is cutting into the ground, revealing its actual geometry is not an amorphous puff but a flat plane.

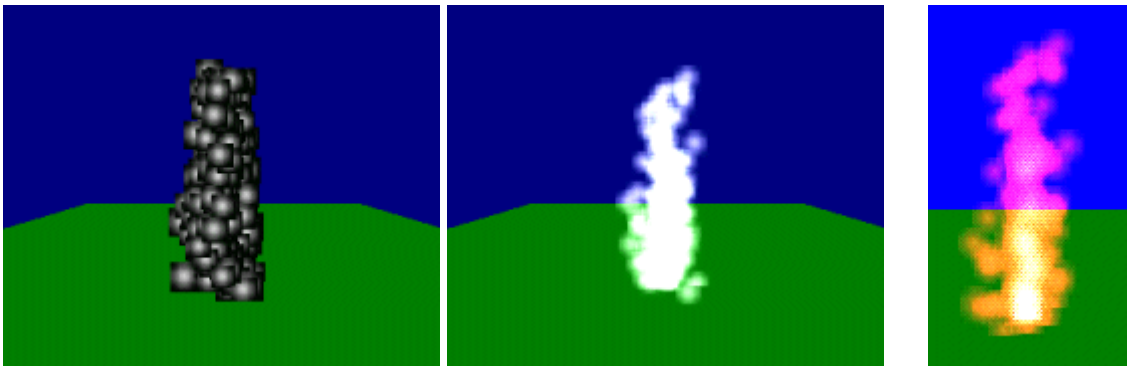


Figure 5.1d) An example of rendering particles using sprites and billboarding. In the left screenshot, the particles are rendered without alpha-blending and simply textured rectangles with a typical particle texture are shown. In the middle and right screenshots, alpha-blending is enabled and source and destination blend factor are both set, color components are also included which tricks our eyes.

5.1.1 Methodology of Billboarding

In general, billboarding involves three main steps: 1) Setup the vertices of sprites, 2) Setup the texture mapped to the billboard, and 3) Setup the modelview matrix relative to the camera's position and orientation. After the above processes, the sprites are updated and rendered in 3D world with respect to the camera.

Step1 : Setup the vertices of sprites

Similar to 3D geometric objects, so-called sprites (also named imposter) is actually a flat plane of 4 vertices with texture mapping rendered in 3D world. The vertices are passed into **vertex buffer** (frame buffer in memory for tracing vertex to vertex) where the vertices will be processed and rendered afterwards.

To define a sprite, 4 vertices with structure(*x, y, z, tu, tv*) are declared and the faces are created in an anti-clockwise order with normals facing towards the camera, where (x, y, z) is the 3D position of the vertices and (tu, tv) are the texture coordinate mapped onto the sprite. In this project demo program, the declaration of sprites in details is as follows :

```
Vertex[a].x = -x
Vertex[a].y = -y/2
Vertex[a].z =  0

Vertex[b].x =  x
Vertex[b].y =  -y/2
Vertex[b].z =  0

Vertex[c].x = x
Vertex[c].y = y/2
Vertex[c].z = 0

Vertex[d].x = -x
Vertex[d].y = y/2
Vertex[d].z = 0
```

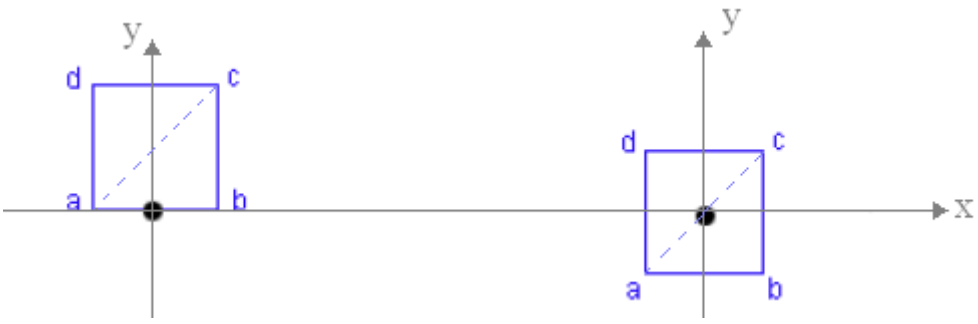
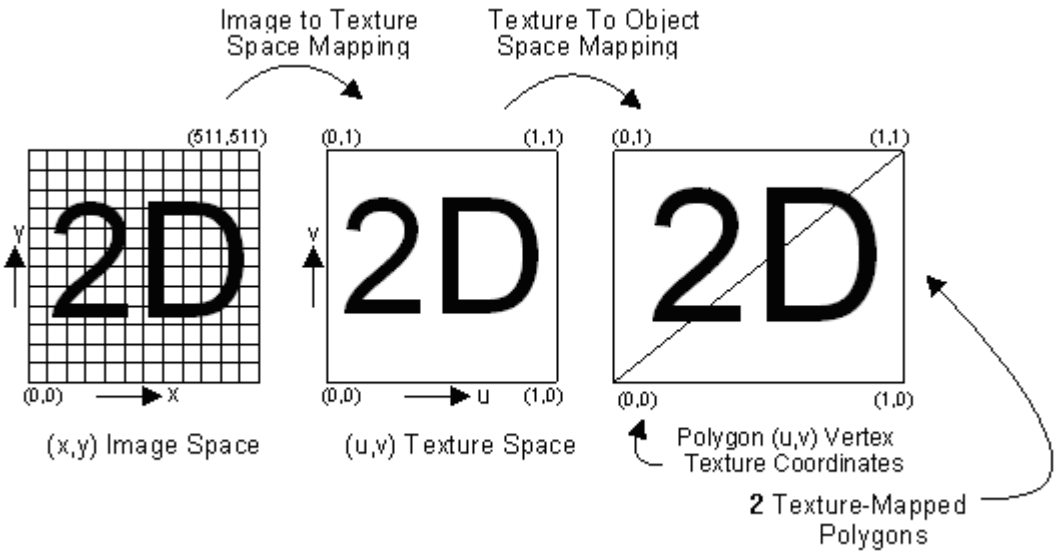


Figure 5.1e) Two examples of a billboard of showing how to define vertices. In the demo program, the structure of the sprites is declared as the right figure.

Step2 : Setup the texture of sprites

For texture mapping of a sprite, a flat 2D image (bmp/jpeg/png format) is mapped onto a flat 3D plane mesh. The following diagram illustrates the process of texture mapping (Figure 5.1f) in which the left diagram represents a 512x512 resolution digitized bitmap image. In the right diagram, it contains 2 triangles which four corners are assigned (*tu,tv*) texture coordinates with ranging [0 1]. The 2D texture mapping process consists of mapping the bitmap pixels of the left diagram to the 3D surface of the polygons and hence the sprite is resulted on the right diagram.



Vertex[a].tu = 0
Vertex[a].tv = 0

Vertex[b].tu = 1
Vertex[b].tv = 0

Vertex[c].tu = 1
Vertex[c].tv = 1

Vertex[d].tu = 0
Vertex[d].tv = 1

Figure 5.1 f) Process and declaration of texture mapping. Notice that the texture is normalized to [0 1] in texture space (i.e. the texture size is mapped in ratio to the whole texture size.)

Step3 : Setup the modelview matrix relative to camera's position and orientation

Before rendering the billboard, it is necessary to identify the position of the billboard which would be displayed on the screen with facing to the eyepoint.

The approach to achieve this requires the modelview matrix. This matrix stores the geometric transformation (rotation, scaling, translation) and it contains the required transformations to change the coordinates inputted, world coordinates, into camera coordinates.

To translate the sprite to the desired position, the computation of the translation matrix is :

$$P' = M \bullet T \bullet P \qquad (5.1A)$$

where P' is the result transformed matrix,
M is the current modelview orthogonal matrix,
T is the translation matrix
P is the matrix with the position of the underlying object representing by this billboard

Without changing the modelview matrix, the vertices of the billboard are manually transformed by reversing the orientations presented in the modelview matrix. That is, by inversing and transposing M , the vectors of the billboard is extracted for the camera to look at.

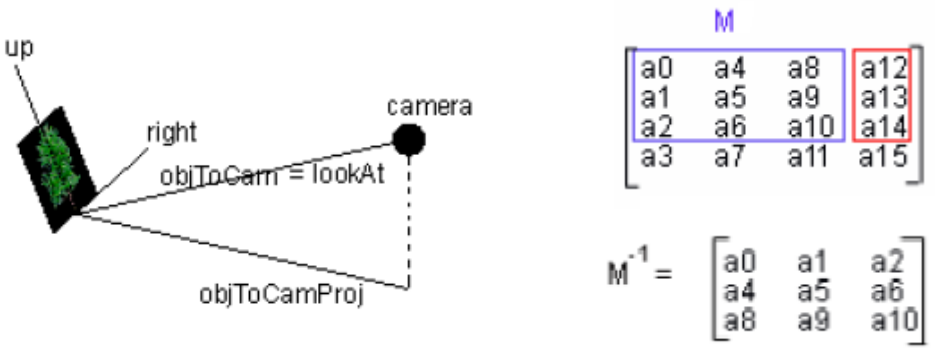


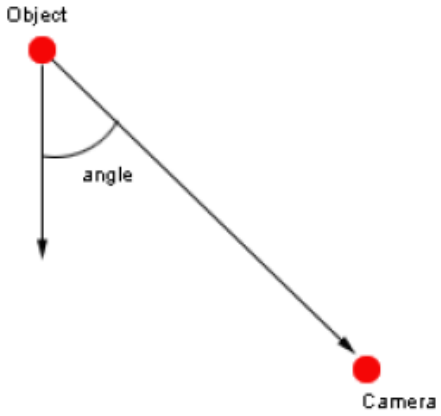
Figure 5.1g) Concept of the relationship between camera and billboard , and the modelview matrix

The vertices of billboard are defined using up and right vectors, supplemented by lookAt vector from the camera:

Right vector = [1, 0, 0]

Up Vector = [0, 1, 0]

LookAt vector = [0 ,0, 1]
(objects is looking along the positive Z axis)



To orientate the object, a rotation is performed around up vector, by the angle between the lookAt vector and the vector from the object to the camera. The required projection is shown in the figure and the vector is defined as :

$$Direction\_Vector = Camera\_Position - Object\_Position \quad (5.1B)$$

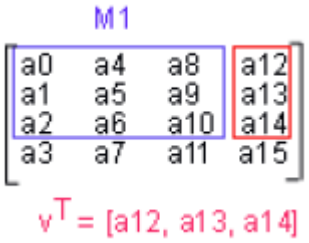
By computing the direction_vector between the camera and the objects, we can easily keep track the required angle for the lookup vector and tilt the objects about Y-axis in order to force the sprites facing to the camera.

Since it is not easy to keep track the world coordinates of the sprites in the world using translations and rotations, we have to do some tricks between the position of sprites and camera. The technique is to reverse the orientation of the camera back to a coordinate system with axis aligned with the world coordinates system, and then the camera position is added to the objects position relative to the camera. This sum will provide the position of the object in world coordinates with the expression of :

$$Object\_Position_{(WC<VC)} = camera\_Position + M1^{T} * V$$

where $M1^T$ is the transpose of the modelview matrix of camera,

V is the vector of the object's position relative to the camera



Note : The above three procedure are processed in game main "Update Class", and the sprite is rendered with refer to the vertices declared by passing appropriate parameters to the game global "Render Class". The parameters of the size of vertex and the loading of texture file should be properly declared and created beforehand in game global "Initialization Class"

```
Setup_billboard_position(float Obj_PosX, float  Obj_PosY,  float Obj_PosZ,
                         float Cam_PosX, float Cam_PosY, float Cam_PosZ){

    // Declare an identity matrix in initialization
     Identity_matrix(Matrix[][]);

     // Translate the billboard into 3d world (object position)
     Matrix[4][1] = Obj_PosX;
     Matrix[4][2] = Obj_PosY;
     Matrix[4][3] = Obj_PosZ;

     // Calculating normal from camera to target character with direction vector
     Direction = (Obj_PosX, Obj_PosY, Obj_PosZ)  –
                 (Cam_PosX, Cam_PosY, Cam_PosZ)


     // billboard self-rotation (about Y) for always facing the camera when camera rotate
     // where RotateY is a defined matrix

     if( Direction > 0 )
         RotateY = Rotate_about_Y( -atanf(Direction.Z / Direction.X) + PI/2 );
     else
         RotateY = Rotate_about_Y( -atanf(Direction.Z / Direction.X) - PI/2 );


     // multiplying matrix (billboard is the required matrix to set up the position in 3D world )
     Billboard = RotateY * Matrix;

}
```

Figure 5.1h} .Psedo-code used for setting up modelview matrix and billboard position in the demo program.

5.1.2 Implementation of Sprite Animation

In a 2D 3D graphics environment, character are rendered in 2D sprite with a series of animation and correlated with the direction facing to the camera in arbitrary angle. The technique is demonstrated as follows.

As stated in the Character Structure (Section 4.2) in a RPG, the direction angle which the character is facing (*character direction*) and the frames of the texture*(tu, tv)* are required for rendering sprite animation. The method is to amend the setting up of the texture used in the billboard (Step 2 of Section 5.1.2) and partition the texture into a smaller cell.

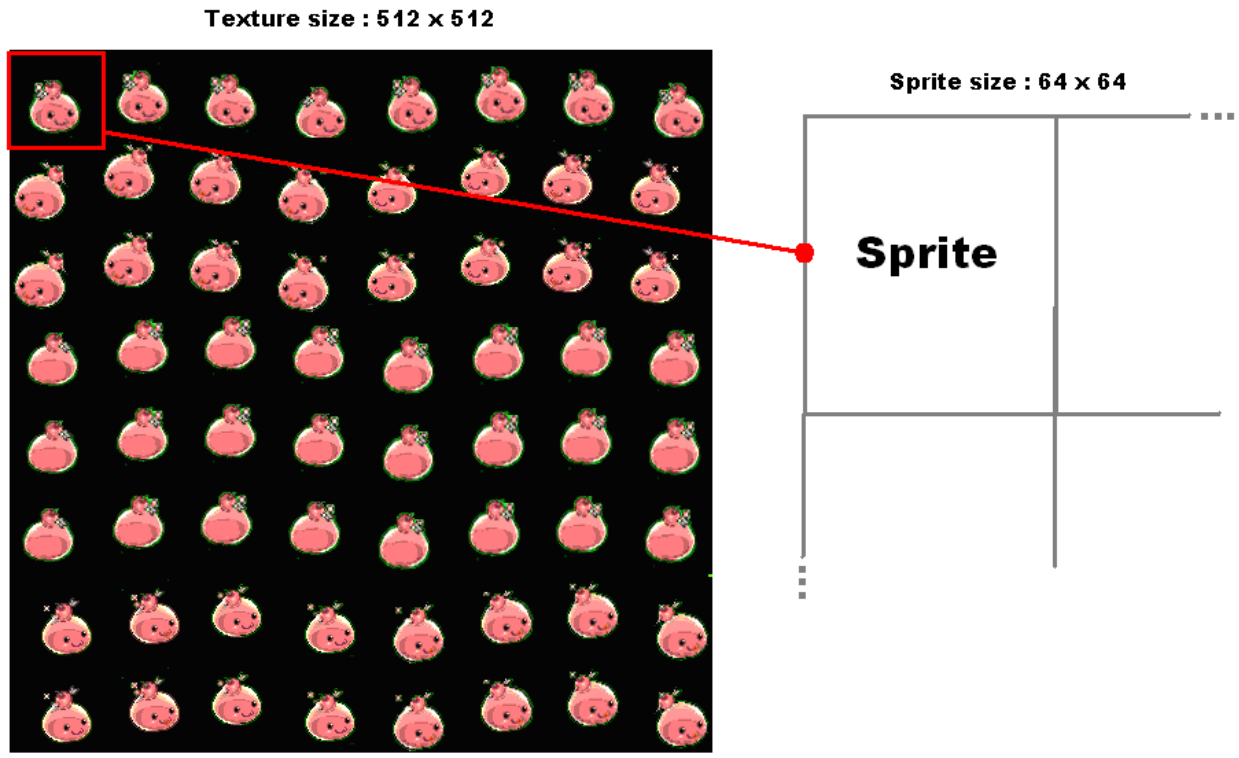


Figure 5.1i) Texture used in the demo program. The whole texture is drawn with a portion of every cells which will be mapped to the screen as sprite animation

Every cell in the figure is called a sprite. According to the game state, the graphic engine selects some of them to be drawn on the screen. Each of the sprites can be copied to the screen individually. In the game, the size of the sprite is declared as 64 x 64 pixel which in terms of slicing the whole texture into 8 parts. The declaration of texture size is arbitrary; however, the performance is better for the size with power of 2. For each sprite to be rendered, the texture coordinate (*tu, tv*) is mapped with sprite size (*SpriteWidth, SpriteHeight*) proportional to whole texture size (*TextureWidth, TextureHeight*), filter out unwanted pixel (using a specified color for

transparent representation) and shift to the next coordinate for animation.

```
float ptX = SpriteWidth / TextureWidth;
float ptY = SpriteHeight / TextureHeight;

float ptX2 = (SpriteWidth + TextureWidth / 8 ) / TextureWidth;
float ptY2 = (SpriteHeight + TextureHeight / 8) /  TextureHeight;

Vertex[a].tu = ptX                    Vertex[c].tu = ptX2
Vertex[a].tv = ptY                    Vertex[c].tv = ptY2

Vertex[b].tu =  ptX2                  Vertex[d].tu = ptX
Vertex[b].tv =  ptY                   Vertex[d].tv = ptY2
```

The purpose of partitioning the texture into a series of animation cells instead of using the whole texture for one animated sprite is tempt to decreases the number of execution of opening and closing texture file. From feasibility studies, opening and closing a data file to the frame buffer introduces overhead to the application program. Although CPU can directly write to the frame buffer of the graphic card, it hurts performance if large numbers of textures are copied from the main memory to the frame buffer for every frame. For instance, a resolution of 800x 600 with 16 bits color depth and 60 frames per second will cost approximately 55MByte/s data flow on the bus link between CPU and graphics card. In optimization, the whole texture is copied to the frame buffer in advance. During rendering, the game engine copies one rectangular region of the frame buffer contents (usually cells of the sprite map) to the viewport (screen) by changing offset of the texture coordinates(tu, tv).

To render the animation sequence, the series of sprites motion is defined in a systematic order with respect to the direction (always defined across the row). Each sprite has its frame number which in terms to be interpreted as a sequence in the program. Starting from the left, frame number ascends and loop back to the first move, it forms an iterative motion with respect to elapsed time. (Figure 5.1j)

Figure 5.1j) Animated sequence of sprite with unique frame number. Notice that every sprite is drawn with iteratively animated movements across the rows.

```
Update2DAnimation(long Frame,  long Elapsed,
                  float XMove, float YMove, float ZMove)
{


   // Update animation frame
   // Change direction of travel (for animation)
   if(XMove > 0.0f || (ZMove) > 0.0f) {          // when character move, change the texture
       Frame += (float)(Elapsed *  0.009);
       if(Frame >= 8.0f)
           Frame = 0.0f;
   } else {
       Frame -= (float)(Elapsed *  0.009);
       if(Frame < 0.0f)
           Frame = 7.0f;
   }
}
```

Figure 5.1k) Psedocode for updating animation sequence of sprite within time.

For defining character direction, the motion of sprites starts from SOUTH, SOUTHWEST, WEST, NORTHWEST, NORTH, NORTHEAST, EAST, to SOUTHEAST throughout the columns. (Figure 5.1i). The process of animating the sprite with different direction is done by shifting the texture coordinates (tu,tv) downwards according to another parameter (Sector number). The value identifies the specific partition to be located on the texture by direction and it is used for determine the width and height of the sprite size. By storing the pixel coordinate (TileX, TileY), the relative sprite is accessed and rendered while processing the character list afterwards.

```
Tile  = Sector * 8 + (long)(Frame);
TileX = Tile % 8 * SpriteWidth;
TileY = Tile / 8 * SpriteHeight;
```

In the project, camera can be rotated in arbitrary angle along y-axis. It is unrealistic that the sprite texture remains facing unchanged if the camera moves. To render this viewing effect, some technique is acquired to the texture mapping with reference to the camera. In the declaration of direction which character facing, the angle starts from South and increase with 45 degree clockwise every turning points. The camera notation is tilt to this direction declaration, where the triggers start between the defined directions. Taking an example, a sprite is facing SOUTH (0^o) while the camera rotates in clockwise direction from SOUTH to SOUTHWEST, the sprite rotates to face SOUTHEAST according to the camera view point when camera triggers the mid-point of the SOUTH direction angle (22.5^o).

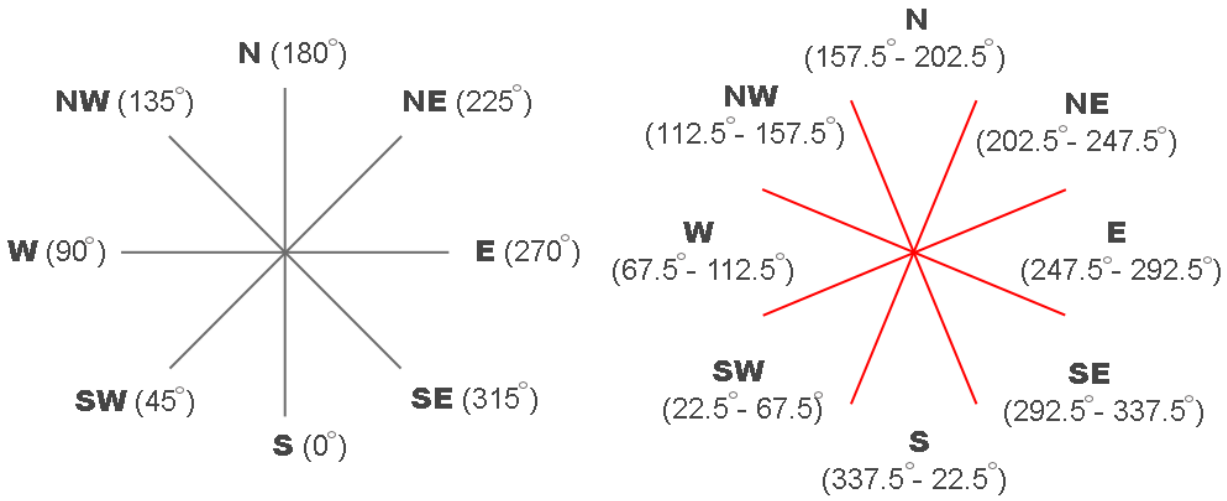


Figure 5.1l) Direction declared for character and camera notation to the direction angle

The implementation depends on the parameter of angle that camera viewing. The definition of the angle is opposite to the direction where camera is viewing. When the camera rotates in clockwise direction, the angle decreased. Conversely, the angle increased. If the direction that character facing is bounded to the angle, the character direction is triggered according to the camera direction.

One potential problem to the above declaration ignores the possibility of character manipulating by rotating the camera. In result, character is not facing towards the correct direction when the camera is changing; even an appropriate movement is made.

The possible solution is adding a pair of viewing protocols. Before updating the character movement, the character direction is updated by:

$$\text{Character Direction} = \text{Character Direction} - \text{Camera Angle} \quad (5.1C)$$

From the formula, character direction is changed accordingly to the camera and it is used for manipulation the character to correct position and facing a correct direction

In updating 2D animation, the actual character direction towards camera is resolved and prepared for rendering. By summating the angle with character facing direction, the resultant angle is the sector required within the 8 directions which in terms of a reverse process of fetching character actual direction. That is, whenever the camera viewing is opposite to the direction of character facing, it is found that the character is always facing to the camera (tempts to facing SOUTH). In calculation, say the character is facing WEST (90^o) and the camera rotates clockwise until facing EAST (270^o), the sum is 360^o which interprets of facing SOUTH and it is deterministic for other cases by the following formula:

$$Resultant\_ViewingAngle = Camera\_Angle + Character\_Direction. \quad (5.1D)$$

```
Update_CharacterMovement ()
{
  ….
   Character Direction  = Character Direction – Camera Angle
}

Update_ViewCamAnimation(sCharacter *CharPtr){

        int sector;

        // Angles in radian (22.5, 67.5, 112.5, 157.5, 202.5, 247.5, 292.5, 337.5)
        float Angles[8] = {0.393f, 1.178f, 1.963f, 2.748f,      \
                                                3.533f, 4.318f, 5.103f, 5.888f};

        // Tracking angles
        // if the major direction angles falls into the critical angle sector,
        // display the sprite with the related direction
        //
        For each 8 direction{

                //Get the angle that camera is facing

                //Calculating the resultant_viewing angle (Reverse process)
                ViewDir  =  Camera_Angle + Character Direction

                if ((ViewDir >= Angles[(i-1)%8]) && (ViewDir < Angles[i%8])){
                        sector = i;
                        break;
                }
                // sector – south (last case)
                sector = 0;
        }

}
```

Figure 5.1m) Psedocode for updating direction of sprite.

5.1.3 Result of Billboarding

Billboarding enables 3D Sprites to be rendered essentially texture mapped 3D facets that always have their surface normal facing into the camera and the Z-order are sorted by default.(Figure 5.1n). Implementation of the technique is done with a variety of possible methods. This paper suggests only one method whilst some research suggests using matrix computation for animation which provides more optimized solution and increase performance.

For creating an effective illusion with the technique:

- the image inside the sprite depicts a three dimensional object
- the animation is constantly changing or depicts rotation
- the sprite exists only for a short period of time
- the depicted object has a similar appearance from many common viewing angles (such as something spherical)
- the viewer accepts that the depicted object only has one perspective. (such as small plants or leaves)

Figure 5.1n) Result of billboarding. In the demo, the sprites is sorted out with painter's algorithm defaulted in 3D graphics representation

5.2 Technique of Mouse Picking

Mouse Picking is a method which selects a specific voxel on the 3D object by clicking a specific point on the screen. This technique is widely used in selecting character or locating world position for the character.

There are many ways to handle mouse picking and specifically 2 ways are highly uses in the commercial game industry : ray tracing and depth buffer testing. Raytracing is extremely accurate but expensive in terms of speed and memory usage. On the other hand, depth buffer testing is fast but its accuracy varies depending on the depth of the z-buffer, so it is not as accurate as raytracing. Since most applications will only perform work when the user actually clicks, the relative expense of raytracing is usually negligible and the technique is preferred. In this paper, the theory behind is applying ray tracing (Figure 5.2a) which projected a ray vector to the 3D environment and compute the corresponding voxel on the 3D surface for manipulation.

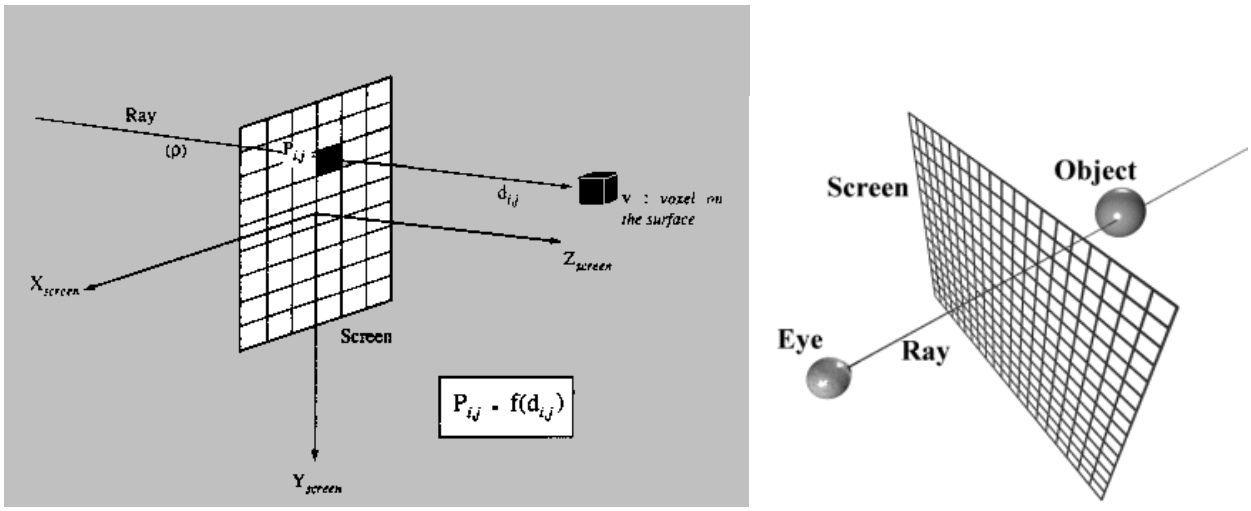


Figure5.2a Principle of Ray tracing on a specific voxel on the surface

5.2.1 Methodology of Mouse Picking

The method is a significantly algorithm used in computer games to acquire three dimensional objects from two dimensional screens by following rays of light from the eye of the observer to a light source using the technique of ray-tracing. By computing directly to the intersection point of a surface on its way from the eye to the source of light, the technique retrieves a normal vector of the specific point based on the ray direction. The high speed of

calculation of ray casting made the mouse picking method perform better in selecting objects in the game for desirable action.

By projecting a geometric ray from some starting point in 3D space, in some direction, and determining where that ray intersects the terrain's surface. For the purposes of mouse picking with ray tracing, we cast the ray from the point on the screen where the user clicked out into the 3D world behind the monitor. If this ray intersects the terrain surface, then we can say the user has clicked that point on the terrain and the value is used for further implementation.

The outline of the basic approach of implementing mouse picking is:

1. The user clicks the mouse (or some input device). Capture the X and Y position of the click.
2. Project a ray from the XY Position.
3. Gather the current view matrix and world matrix of the 3D environment to distinguish the intersecting points between the terrain mesh and the ray.
4. Checking the intersection of the ray.
5. Get the resultant vector of the specific position on the mesh, if any.

```
If (mouse button is clicked)
{
        long cursorX = Mouse.XPos();
        long cursorY = Mouse.YPos();


        // Convert from screen to view space:
        // (Compute the vector of the pick ray in screen space)
        // vRay (the line of ray to be projected : temporary vector)
        D3DXVECTOR3 vRay;
        vRay.x =  ((( 2.0f * cursorX ) / WindowsWidth() ) - 1.0f ) / matProj._11;
        vRay.y = -((( 2.0f * cursorY ) /WindowsHeight() ) - 1.0f ) / matProj._22;
        vRay.z =  1.0f;

        // Transform the screen space Pick ray into 3D space
        // vRayDir  (the direction of ray)
        // vRayFrom (the origin position of ray)
        D3DXVECTOR3 vRayDir, vRayFrom;
        D3DXMatrixInverse(&matInv, NULL, &matView);        // Get the inverse view matrix
        vRayDir.x = vRay.x * matInv._11 + vRay.y * matInv._21 + vRay.z * matInv._31;
        vRayDir.y = vRay.x * matInv._12 + vRay.y * matInv._22 + vRay.z * matInv._32;
        vRayDir.z = vRay.x * matInv._13 + vRay.y * matInv._23 + vRay.z * matInv._33;
        vRayFrom.x = matInv._41;
        vRayFrom.y = matInv._42;
        vRayFrom.z = matInv._43;

        // intersect with terrain and return the position on the terrain
        if  (check Intersection with terrain)
            TargetPosition = vRayFrom + vRayDir * Distance;
        }
    }
```

Figure5.2a Psedocode of Ray Casting using DirectX9.0

5.2.2 Result of Mouse Picking

To manipulate the movement for user using the Mouse Picking technique, the resultant vector of intersection is used for calculation. (Figure 5.2c) By computing the difference between player's position and its target position, the orientation between the player and the target position is acquired and the players decrease its movement towards the direction every elapsed time until it reached the final destination.

```
// if position is set, process auto move to target position
if (not reached to the destination && target position is set){

        // check if the target position is not reached, calculate the orientation
        float XDiff, ZDiff, Dist;
        XDiff = TargetPos.x – CharacterPos.x;
        ZDiff =  TargetPos.z – CharacterPos.z;
        Dist = XDiff*XDiff + ZDiff*ZDiff;

        if (Dist > 100){
                // movement  continue
        }else{
                // stop movement
                // reset target position to the status "not set"
        }
}
```

Figure5.2b Psedocode of Character move towards the target position

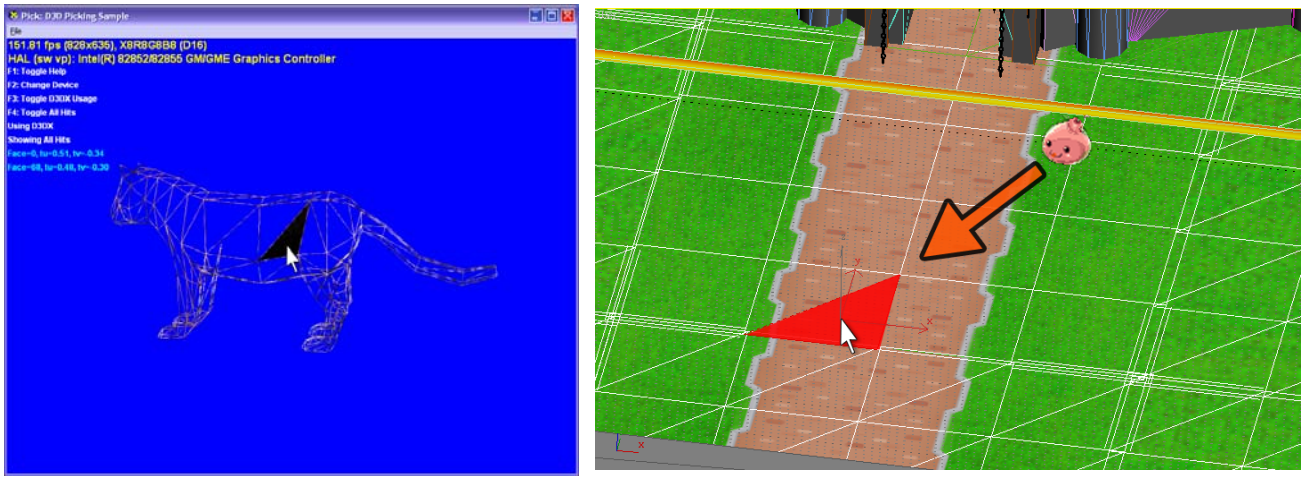


Figure5.2c An example of selecting polygon using mouse picking. In the left screen, it shows how the selected triangle from a wire-framed mesh is found by ray tracing and its value is acquired. In the right frame, it demonstrates the concept of ray tracing on the land for the application of point-and-go system.

6. Machine Learning in Game Development

In general, machine learning involves adaptive mechanisms that enable computers to learn from experience, learn by example and learn by analog. With better learning capabilities, the performance of an intelligent system can be improved over time. Therefore, machine learning mechanisms form the basis for adaptive systems and the most popular approaches to machine learning are Neural Networks and genetic algorithm.

Although there has been much discussion in the game development community of the desirability of in-game learning and adaptation, there have been few commercially successful implementations in such capabilities. One possible reason is the techniques of neural networks and genetic algorithms are usually least suitable for producing these effects due to high cost and computational inefficiency, as well as players train their opponents are less preferred. The technique potentially can be used to adapt as the game is played, but this is a rather intriguing possibility for solely use. Therefore, some games introduce the use of combination of other AI algorithm approach supplemented with Neural Network in order to stabilize the result. Good examples such as dynamic scripting mixed with neural network or the combination of decision tree and neural network are the techniques for reducing the size and complexity of search space, which it allows AI system to explore the search space quickly and efficiently in the game. In this project, **Neural Network** is dedicated and the adaptive AI is trained with an approach of preference-based player modeling.

Figure6. The game, *Black & White*, by Lionhead Studio, included adaptive AI of dynamic scripting and neural network, allowing the player to guide the creatures as a teacher to do substantial mission. [4]

In more practical terms, the Neural Network system is capable of ‘learning’ which use experience to improve its performance. Application ranged from recognition, identification to imitation. In game development, before the games are being shipped, one possible use of the AI is as non-linear statistical data modeling tools which used to find pattern between inputs and outputs to model a complex relationship. That is, game developers use the AI for coding optimization and break down the complex state machines or rules-based systems by relegating key decision-making processes to one or more trained neural networks.

6.1 Introduction of Neural Network

In terms of information processing, neural network is a system that approximates the operation of human brain with a set of mathematical function. It consists of a number of interconnected processors (neurons) which resemble to the neurons in the brain and they are connected by weighed links passing signals from one neuron to another. After receiving a number of input signals, the system then transmits the output signal through the neuron’s outgoing connection and the outputted signal is used for further determination and decision making. Fundamentally, the neurons are actually data with a mathematical function to help adjust their output value from current input value.

Neural Network is diversified to many types such as single-layer perception network, Hopfield network, stochastic network, Boltzmann machine and others. Among them, multi-layer perception neural network is applied in this project because of their flexibility to solve a wide range of practical problems. In order to reduce the cost of long and computationally intensive training, less perception is used in the system for in-game training.

Pictorial Description of Neural Network

A general structure of Multi-Layer Perception Neural Network is shown in Fig. 6.1. The neural network consists of 3 main nodes: **Input Layer,** at least one **Hidden Layer** and **Output Layer**. Normally the information moves in forward direction, from the input nodes, through the hidden nodes and to the output nodes. The nodes are connected by links and each link associated with a numerical interconnected weighs calculated for learning purpose. Input to the network are the independent variables which are always some criteria based on the users' input, while the output represents the dependant variable relied on the inputted variables and by the hidden layer itself. In other words, the network itself is a function giving one unique set of output for the given input. This mapping is highly nonlinear which depends on the structure of how the adaptive AI system is being trained.

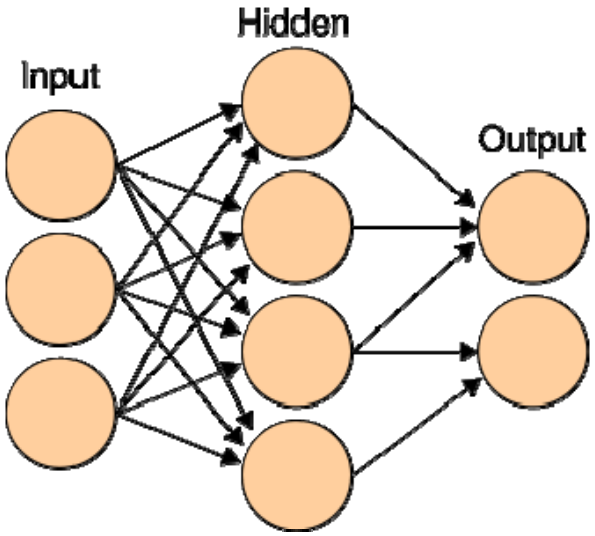


Figure 6.1 Typical ANN illustrates the interconnected groups of nodes in the system

6.2 Learning Procedure of NN

In definition, learning interprets the process of building up memory based on the stimulation by the environment. In ANN, the interconnection weights are the basic means of long term memory. By modifying the associated weights of the interconnections inside the ANN, the network 'learns' through the iterative adjustments of these weights.

The means of a neuron to learn and determine its output is by computing the weighted sum of the input signals. Each input would be scaled to vary roughly in the range [0, 1] so those that normally vary over a greater range do not dominate the network's calculation. For a single-layer neural network, the inputs connect directly to their outputs without hidden neurons and hence the output is usually computed as a linear superposition of its inputs:

$$y = \sum_{n=1}^{N} w_n x_n \qquad (6.2A)$$

where x_n are the values of the network's N inputs,
w_n are the networks ,
y is the network's output

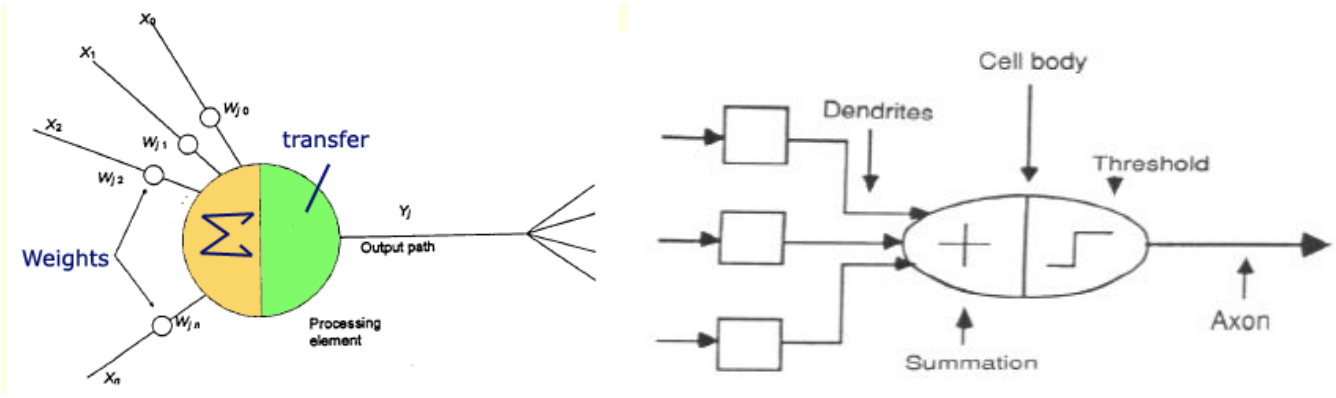


Figure 6.2 Diagram of a multi-layer neuron, showing the computation of output by summating interconnection weights and multiply with a transfer function

On the other hand, the output of a multi-layer neural network based on the weighted sum of the input with comparison to a step size parameter that controls the trade-off between the speed of the learning and its sensitivity to noise, typically has a ranged value of [0.01 0.1].

That is, the neuron uses the following step activation function with a threshold value, θ to attain the output value, where θ is used to shift the decision boundary in the search space as a learning rate.

$$X = \sum_{i-1}^{n} x_i \, w_i$$

$$Y = \begin{cases} +1 & \text{if } X >= \theta \\ -1 & \text{if } X < \theta \end{cases} \qquad (6.2B)$$

where X is the net weighted input
x_i is the value of input of input i
w_i is the weight of input i
Y is the output

Since the network is used to estimate probabilities, the output is often calculated with a sigmoid activation function (Equation 6.2C) instead of a step function so it is guaranteed to lie in the range [0, 1] and used for classification and pattern recognition tasks.

$$Y^{\text{sigmoid}} = \frac{1}{1+e^{-x}} \qquad (6.2C)$$

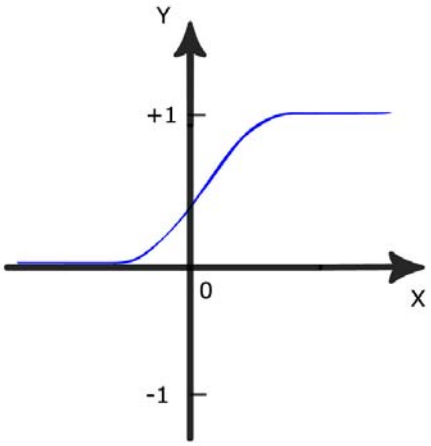


Figure 6.2.1 graphical representation of a sigmoid activation function

In this case, each interconnection weight is updated with a set of reasonable functions and the system ‘learn’ through this iteration process with reasonable outcome.

The mechanisms of classifying tasks in typically ANN consists of 4 parts : **Initialization, Activation** (Simulation), **Learning** (Weight training) and **Iteration**. Developers initiate the neural system by designing the network architecture, and then decide which learning algorithm should be applied. And finally initialize and update the weights of the network to train the system from a set of examples.

Step 1 : Initialization

Typically, the adaptive AI is initially set all the weights w_1, w_2, … w_n and threshold value θ of the network to random numbers uniformly distributed in a small range. After processing for one iteration, usually the system is fed with a set of information which is the data and rules about the data relationship, so called **Training Set.** Akin to a recipe which guide users what to do and what would the expected results, a training set consists of a combination of the provided input and corresponding desired output as illustrated in Figure 6.2.2 which is manually hard-coded or extracted from player model.

```
double          TrainingSet[14][7] = {
     0,     1,     0,     0.2,    0.9,    0.1,    0.1,
     0,     1,     1,     0.2,    0.9,    0.1,    0.1,
     0,     1,     0,     0.8,    0.1,    0.1,    0.1,
     0.1,   0.5,   0,     0.2,    0.9,    0.1,    0.1,
     0,     0.25,  1,     0.5,    0.1,    0.9,    0.1,
     0,     0.2,   1,     0.2,    0.1,    0.1,    0.9,
     0.3,   0.2,   0,     0.2,    0.9,    0.1,    0.1,
     0,     0.2,   0,     0.3,    0.1,    0.9,    0.1,
     0,     1,     0,     0.2,    0.1,    0.9,    0.1,
     0,     1,     1,     0.6,    0.1,    0.1,    0.1,
     0,     1,     0,     0.8,    0.1,    0.9,    0.1,
     0.1,   0.2,   0,     0.2,    0.1,    0.1,    0.9,
     0,     0.25,  1,     0.5,    0.1,    0.1,    0.9,
     0,     0.6,   0,     0.2,    0.1,    0.1,    0.9
};
```

4 inputs

3 desired outputs

Figure 6.2.2 the code demonstrates the training set in the demo program

In the above training set, it shows a number of data set ranged from 0 to 1 to illustrate a neural system with 4 inputs and 3 desired outputs, depending on the developers to decide the numbers of neurons to be used for the network architecture. This set of data is used for compute the interconnection weights for further adjustments.

Step 2 : Activation

Once the training set is prepared, the system is triggered by applying the inputs $\mathbf{x_1}$ **(p),** $\mathbf{x_2}$**(p), …** $\mathbf{x_n}$**(p)** and desired outputs $\mathbf{t_1}$ **(p),** $\mathbf{t_2}$**(p) …** $\mathbf{t_n}$**(p)** in order to calculate the actual outputs of the neurons in the output layer with the value of interconnection weights by the technique of **feed-forward** (i.e. the information moves in forward direction, from the input nodes, through the hidden nodes and to the output nodes). While simulating, the current inputs from the environmental criteria are fed into the system. The input value is then computed with the activation functions mentioned and finally the actual output is determined with maxima stimulus for every epoch. (Figure 6.2.3)

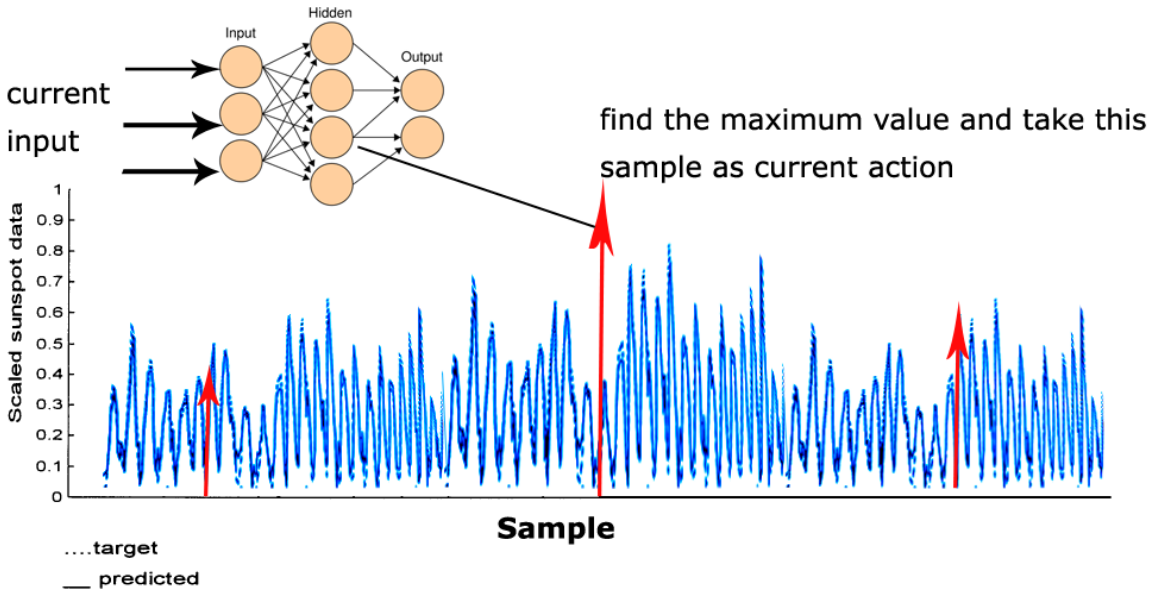


Figure 6.2.3 A diagram illustrates the underlying concepts of actual output determination with respect to the current input approximates to the value from training set sample

Step 3 : Learning

Learning requires updating the weights of the perception. By adding the current weights and the corrected weights computed by the delta rule, each the interconnection links is updating iteratively to train the system. The corresponding formula is calculated as below: (Equation 6.2D)

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n)$$

(6.2D)

If the network actually outputs a value y but an accurate prediction should have been t, each weight is updated according to the equation 6.2E. Since the system undergoes **error-correction learning**, the error signal is counted which is defined as the difference between the desired response $t_j(n)$ and the actual response $y_j(n)$ of a neuron at time n. (Equation 6.2F)

$$w_n = w_n + \alpha \times x(n) \times e(n) \quad (6.2E)$$

where α is the learning rate,

x(n) is the inputs of the perception

e(n) is the error signal

$$e_j(n) = [t_j(n) - y_j(n)]$$

$$\text{where } y_j(n) = f\left[\sum_{i=1}^{N} w_{ji}(n)\, x_i(n)\right] \quad (6.2F)$$

With the technique of **back-propagation** (i.e. the information moves in backward direction, starting from the output nodes, through the hidden nodes and to back the input nodes), the interconnection weights are updated and are being trained for each iteration. After that, the network increases the **iteration** n by one and repeats the process until the selected error criterion is satisfied. In result, the system can be trained to be intelligent.

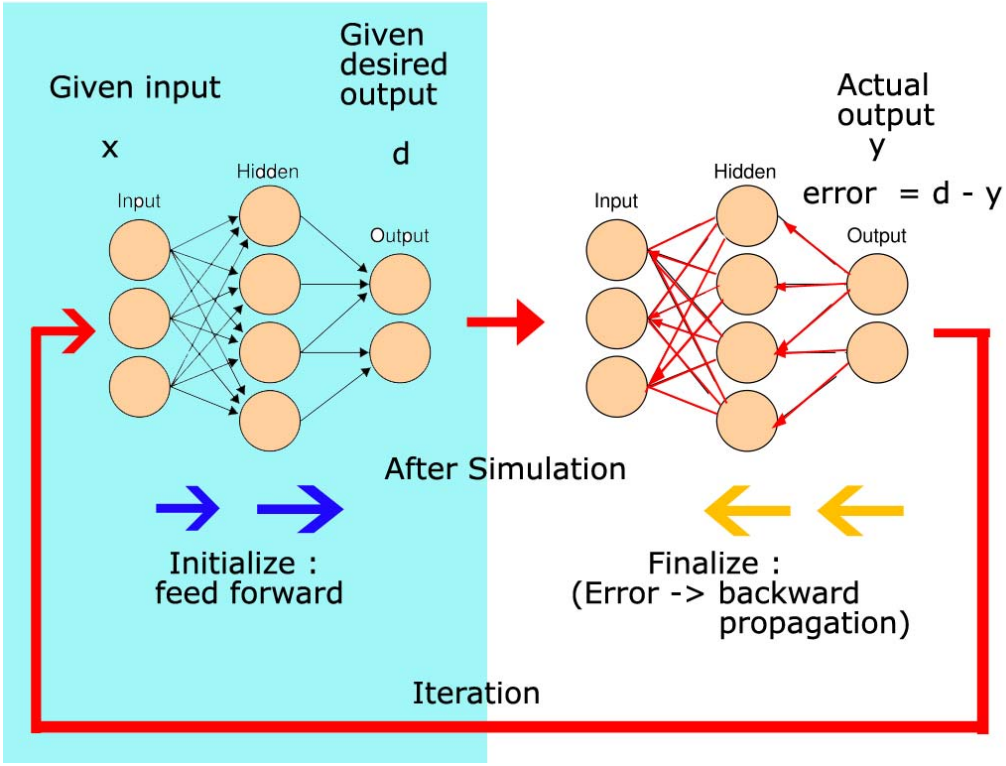


Figure 6.2.4 the diagram showing the mechanism of ANN.

6.3 Methodology of Modeling for non-playing agents

Since game AI highly depends on the game criteria implemented for its game play, sophisticated portal system should be well-defined to enable non-playing agents collects a wide range of attributes for computation. Interesting manoeuvres and behaviour can be learnt through a highly detailed input with a certain complexity and it is the desirability of game developers to work on. To be simplified, in this project, 4 inputs, one hidden layer with 3 computational neurons and 3 outputs are applied in the neural network for online learning and the methodology is explained as below.

Intention and Play

The AI is to be used to estimate the probability that combat between an NPC and a player for the determination of the result in victory or defeat for the NPC. When the encounter starts, the AI would use the network to estimate the probability that the NPC will win. When the encounter ends, the predicted value should be set to 0 if the NPC lost. After several tens of weight updates, the network should provide a reasonable estimate of the probability of the NPC winning any particular encounter.

Regarding to the persistent role-playing game with the subgenre of action type, the neural network can be applied to determine whether the action of attack or flee taken by the non-playing agents in order to control how certain creatures in the game behave. The adaptive AI is used to handle the decision-making process of the creature ranged from attack, evade, or wander, depending on some criteria between the NPC attributes, their enemy‘s(a player) attributes and environmental signal.

Modeling the NPC

To model the non-playing agents, four major inputs are introduced in this project: whether or not the enemy is within the AI agents' proximity, a ratio of the creature's damage value to the health points, a time checking to indicate whether or not the enemy is currently engaged in combat with another creature; and the distance between the enemy and the creature. After the process, the monster switches within 3 critical outputs: wander around, follow the enemy or evade from the enemy. (Figure 6.3)

During game play, the neural network can be used to evolve using a sort of positive or negative reinforcement training. If the creature takes an action suggested by the network and its death results, then it was assumed as a bad decision. In this case, the neural network is trained and adjusted to suppress that decision given the particular set of input conditions. Similar approach can be applied to the events in decision like collision detection and strategy of attacking players, which the events require some additional inputs of environmental criterion.

Input:

- distance between player and monsters
- monsters' HP
- number of monsters around player
- action that the monster is currently taken

Output:

- monster hit player and stop follow
- monster evade when HP < 0.3%
- monster follow player if there is no monster around him

Figure 6.3 a list of rules showing the condition implemented as the input and output in this project

In advance, the adaptive AI can be learn better and little more sophisticated by adding more inputs such as the class of the enemy. In a typical role-playing game, players assume the roles of character with 3 different types of classes: fighter, priest and mage. Neural network can determine the player role of whether or not the enemy is a mage, priest or fighter. This consideration is important to a creature in order to find better suited attack strategies and defenses against one type of class or another, instead of determining the enemy's class achieved by "cheating". With the aid of Bayesian analysis, the enemy's class can be predicted for a better approach, adding a bit more uncertainty to the whole process and increase the attractiveness of the game play.

6.4 Imitation Learning using NN

Theoretically, the trained behaviour of the “intelligent” agent can be obtained from player in order to show the ability of learning interesting manoeuvres and imitate player’s actions. In general, a player model is an abstract representation of certain characteristics of a player and his behaviour can be implemented with the aid of Neural Network. NPC can learn through how players respond to the situations they encounter depending on a variety of factors. If the player’s preferences can be able to predicted for different responses to different situations, it could use the model to adapt to the player, imitating the player when the agents encounters the similar situations, perhaps taking preemptive action to prevent the player repeatedly pursuing the same strategy. [4]

One possible idea is to twist the training set of the learning agents with the guidance of developers (players) before the game is shipped. With the implementation of AI controller class, each character is assigned with a set of rules which can be used individually. In the AI test bench, developers (players) can play with the AI opponents and a training set is generated within a certain time. The data consists of the current action of players, the range of the enemy in vectors, signals from dynamic obstacles (collision detection) and the attributes of players (HP ratio, MP ratio) is then fed as the training set for the AI opponents. By using a correct player model which contains a wide range of relevant features, the learning agents can be tutored to imitate the movements of players and possible actions in a certain situation. The purpose is to allow NPC learns through swapping of roles between players and the AI opponents.

7. Discussion and Evaluation

Problems encountered in the game development are usually from the startup of game programming. Although developing 3D graphics using DirectX is simpler than before, it requires intensive documentation reading which is difficult for newcomers to start up with. In addition, the concept of using OOP technique and the startup of window programming with Win32 API is not easy in the beginning. In order to cope with this sort of problems, it is advised to draw a simple UML showing how the game objects and classes which are triggered within the game loop and newcomers should familiarize with this type of program flows.

In the implementation of AI, some difficulties are also found as the result is not desirable and unstable. In order to breakdown the problem, experimental set-up concerning on the neural system is conducted. For the experimental testing, more than three hidden layer is used to investigate the outcome. In results, the performance of the AI agents rises as smaller numbers of hidden layer is used which prompts to allow system take advantage of direct mapping and more memory efficient. However, decrease in the hidden layers show no desirable outcomes. Conversely, by increasing the number of hidden layer, the results becomes more undesirable and it concludes that the number of hidden layer do no improvement to the network.

With regarding to stabilize the results, momentum should be added in the weight correction process. (Equation 7.0) In this case, the system is improved with a better result evaluated.

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{7.0}$$

Even though the implementation of the current AI system is not success in this demo, the idea behind suggests the possibility of applying Neural Network to imitate player's action. To enable high flexibility in the in-game training, the program should be adjusted so that it allows collecting players attributes while playing. The intention is to train an adaptive AI which can change this action and provide a higher challenge in gameplay,

8. Game Design and Project Implementation

For the progress of the project, a game application is made. The game is operated with a RPG mode which features on physical attacks, magical attacks and heals in a portal system with a set of monsters to combat. To increase the playability, time pressure and greater challenge is introduced. Player can either group together with NPC for a battle in taking missions whereas the aim of the game is to complete the objectives as possible. The basic winning mission is to defeat all the monsters in an enclosed area within time limit.

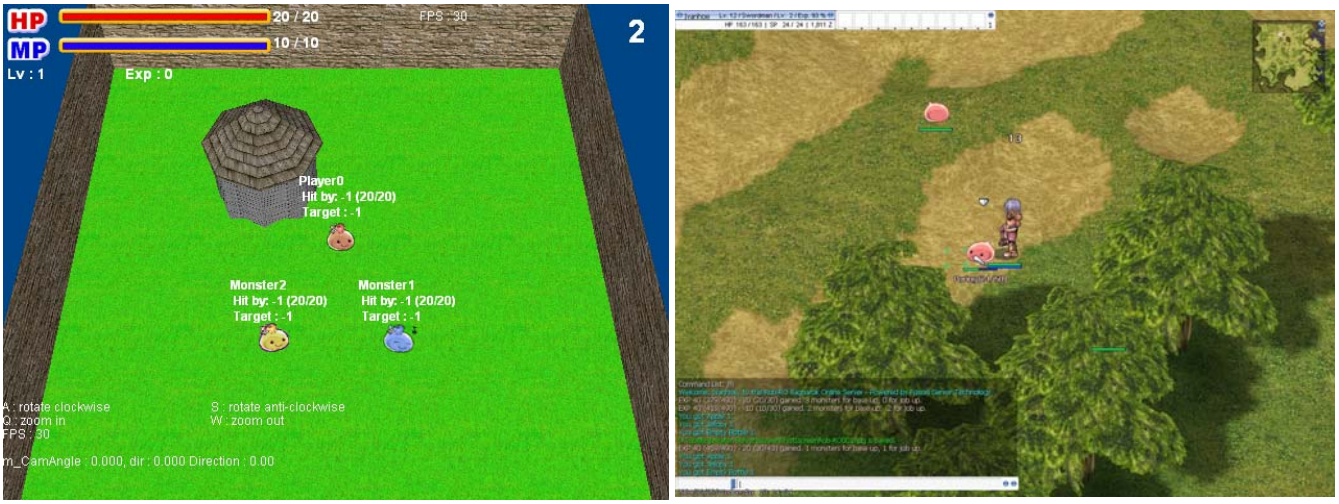


Figure 8 Game Design and the Interface. The left is the screenshot of the demo program and the right interface is the screenshot of a popular commercial MMORPG, Ragnarok Online, presented by Gravity Coporation 2003.

In this game, players are able to control their pinky character (the “slime” – which is a symbolic creature in RPG) and take a combat with other AI opponents within a small restricted area. For the clearly demonstration of AI in the game, the game is emphasis with its action type ability, which is similar to some games with tournament types. The main features of the combat game consist of:

1) A numbers of 4 characters spawn around and attack with each other.
2) Without using directly point to the enemy, players can switch the target enemies by locking them with one press
3) Players are allow to team with AI partner to against other two opponents

Basic Input setting

The following is the key input for players to manipulate their characters:

Keypad	Function
Key_UP	Move forward
Key_DOWN	Move downward
Key_RIGHT	Move to the right
Key_LEFT	Move to the left
Key_UP	Move forward
Key_Z	melee attack
Key_X	Jump
Key_C	Cast spell
Key_V	Lock the target
Key_Esc	Exit the game

Game Prototype Interface

The game is developed into a 3D environment which the scene and model is using X file format. To demonstrate the game with its feature, first person perspective view is constructed by adjusting the camera position and orientation behind the player. Additional graphics features are also applied in the project: Skybox, Camera occlusion, Lighting, Particle System for visual enhancement and further development. In the game prototype, Health Points and Mana Points of the player are displayed on the top left-handed corner with a time counter in text. FPS counter is used to illustrate the number of frame within 1 second for the counting of game performance. Finally, some text displayed on the top of each character is used for debug and as an indicator for player to know about the status of NPC.

Figure 8.1 Snapshot showing the battle of the game

Figure 8.2 A screenshot showing the first person perspective view of the game with gorgeous background settings. The scene demonstrates some CG techniques such as Skybox and camera occlusion.

9. Conclusion and Future Development

In this report, an introduction to game architecture and the methodology of billboard is given. Then, the concept of billboard and sprites are explained. In the further progress, artificial intelligence implementation using Neural Network is illustrated supplemented with pictorial description, followed by giving mathematical formula with evaluation and optimization. Finally, the underlying concept of how to implement more intelligent system with the use of Neural Network by imitation is suggested and the performance of system is briefly illustrated.

For further development, there should be more functions and components built in order to ensure the high quality and flexibility in game implementation.

1) Tools and Editors

A wide range of editors help to enhance the process of game development and lessen the burden for game developers. Character Editors should be implemented so that the in-game character list is not hard-coded and flexible for editing through a more user-friendly interface. Others tools such as effect editors, map editors, script editors and event editors are desirable. In addition, an AI editor with test bench can also ease the workload for developers for creating more intelligent AI agents.

2) Item system and more ammunition

Increasing a wide range of items enhance game mechanisms with good gameplay. Moreover, interesting AI can be implemented with this high range of possible inputs of items.

3) Profiler and Game Optimization

Game Performance is always one of the concerns for developers. With the aid of a profiler, unnecessary graphical burden can be reduced. Some possible ways of tweaking the game speed are applying BSP-tree, decreasing the number of polygons in 3D models, or using less resolution of textures where necessary.

4) Others

Game demands more variety of artistic characters, architecture and environment. Increasing the quality of computer graphics guarantee a high visual improvement in the game as well as encouraging gameplay. In addition, some specific actions and effects should also be implemented in order to attract players with good input supporting system and corresponding kinematics applied in the game.

References:

Books:
[1] Programming Role-playing Game with DirectX / Jim Adams
[2] AI Game Programming Wisdom / Steve Rabin.
[3] AI Game Programming Wisdom II / Steve Rabin.
[4] AI Game Programming Wisdom III/ Steve Rabin

Online resources:
[4] Game Development Forum
http://www.gamedev.net

[5] Sprite Computer Games
http://en.wikipedia.org/wiki/Sprite_(computer_graphics)

Paper:
[6] Image-based rendering for real-time applications
Matthias Baur

[7] Evaluation of Pointing Techniques for Ray Casting Selection in Virtual Environments
Sangyoon Lee, Jinseok Seo, Gerard Jounghyun Kim, Chan-Mo Park