Advanced Technologies Report - RTS AI

Ricardo Heath
15000805

University of the West of England

July 17, 2019

The task is to create an RTS AI within a framework that uses a core graphics API. Direct3D 11 was chosen as the graphics API for a 3D prototype that showcases RTS AI in the form of squad-based groupings and pathfinding.

1 Introduction

The framework will be created using Direct3D 11 for drawing functionality and Win32 for windowing functionality. These API's will form the core of a 3D framework that supports instancing, deferred directional lighting, vertex and pixel shaders, texturing, and camera functionality. A 3D terrain is generated for the world that the RTS AI will occupy and A* will be used as the pathfinding algorithm for units in squads.

2 Related Work

Pathfinding in computer games as described by Cui and Shi, 2011 is done through the use of the A* pathfinding algorithm, which is similar to Djisktras's algorithm with the addition of a heuristic. They go on to explain how a small amount of heuristic overestimation may result in improved performance due to fewer nodes being searched, though the amount by which to overestimate is not known before it devolves to a worse implementation.

The documentation on MSDN (Kennedy and Sartran, 2018) goes in to detail regarding how to use Direct3D11 to create a 3D framework with the rendering pipeline and the COM model. The first ever game to use Direct3D11 was Battleforge (Plunkett, 2009), a hybrid of an RTS and card game. This confirms that my initial approach to the problem is along the rights lines.

3 Method

3.1 Direct3D11

3.1.1 Hardware Interface

Direct3D provides access to the GPU through ID3D11Device and drawing through ID3D11DeviceContext. ID3D11Device is used to configure Direct3D resources prior to rendering while ID3D11DeviceContext is used to load Direct3D resources and perform draw calls that go through our rendering pipeline.

3.1.2 DXGI Adapter

The DirectX Graphics Interface adapter is used to represent a display. The adapter finds the display mode that matches our required window size.

3.1.3 Device & Swapchain

Swapchains are used to enable buffered rendering. This is done by drawing to a buffer while another buffer is displayed to the user and then swapping them.

Once the swapchain description is configured to match our window the swapchain and device can be created together.

3.1.4 Render Target

A render target is used as a surface for drawing pixels to. An ID3D11Texture2D object is created and assigned

as the backbuffer for the swapchain.

3.1.5 Depth Buffer

A depth-stencil buffer is a ID3D11Texture2D object that is used to determine which pixels should be drawn based on how far an object is from the camera. A standard depth-stencil state was described using D3D11_DEPTH_STENCIL_DESC and a depth-stencil view was created as required to enable depth testing.

3.1.6 Rasterizer State

The rasterizer state affects how the rasterization stage converts vertices to pixels. This involves clipping vertices outside of the view frustram and invoking the pixel shader. A normal solid-fill state and a wireframe state are created with back-face culling.

3.1.7 Viewport

The viewport determines the content that will be displayed in the window through the back buffer based on the described rectangle. The full client area of the window is used.

3.1.8 Graphics Pipeline

The graphics pipeline is the process in which draw calls are executed with the input data processed such that pixels are generated and output to the render target. Figure 1 shows the Direct3D11 Pipeline. This project uses the input-assembler, vertex shader, rasterizer, pixel shader and output-merger stages of the pipeline.

3.1.9 Matrices

World, projection, and view matrices are created and used to determine the transforms of objects relative to the screen and the game world by converting between different spaces. A standard projection matrix is created based on the window size. The view matrix is constructed by the camera and the world matrix is constructed within the Vertex Shader.

3.1.10 Shader Language

Dierct3D uses the HLSL (High Level Shader Language) to program shaders, which is similar to C. HLSL Shaders can be compiled to CSO (Compiled Shader Object) files at compile time, or compiled at runtime from an array of characters. This project compiles them at runtime.

3.1.11 Vertex Shader

The vertex shader is used to convert vertices to pixel based mappings. These vertices are manipulated by the world, view, and projection matrices.

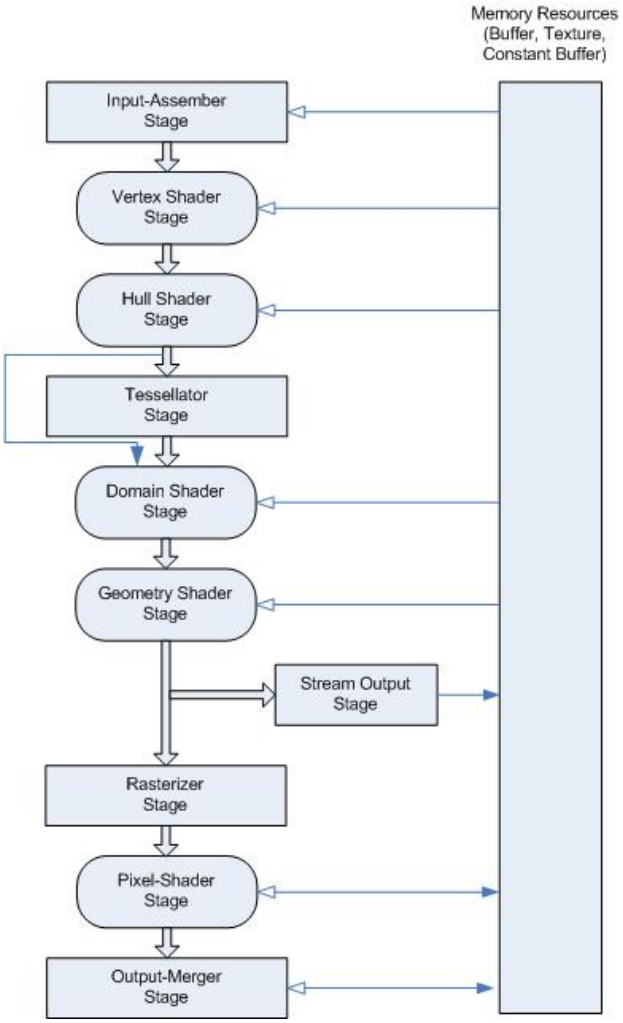

Figure 1: *The entire Direct3D11 Graphics Pipeline as described by Microsoft*

This projects vertex shader creates a world matrix based on the rotation, scale, and position of an instance.

$$Scale = \begin{pmatrix} scale_x & 0 & 0 & 0 \\ 0 & scale_y & 0 & 0 \\ 0 & 0 & scale_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1)$$

$$RotationX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(rot_x) & -sin(rot_x) & 0 \\ 0 & sin(rot_x) & cos(rot_x) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2)$$

$$RotationY = \begin{pmatrix} cos(rot_y) & 0 & sin(rot_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(rot_y) & 0 & cos(rot_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3)$$

$$RotationZ = \begin{pmatrix} cos(rot_z) & -sin(rot_z) & 0 & 0 \\ sin(rot_z) & cos(rot_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4)$$

$$Rotation = RotationX \times RotationY \times RotationZ \quad (5)$$

$$Position = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ pos_x & pos_y & pos_z & 1 \end{pmatrix} \quad (6)$$

$$World = Scale \times Rotation \times Position \quad (7)$$

The world matrix is applied to the vertices, followed by the view and projection matrices. In a non-instanced shader the world matrix would be passed using a constant buffer. The order of matrix multiplication is important as they are non-commutative.

The vertex shader outputs are fed to the pixel shader.

3.1.12 Pixel Shader

The pixel shader stage is used to determine the final colour of a pixel. The output data from the vertex shader stage is used to calculate lighting data and texture sampling to determine the correct pixel colour.

Lighting consists of ambient light and directional light. The directional light information is passed to the pixel shader from the CPU. The lighting in this project is diffuse.

Diffuse lighting estimates the impact that a light has on an object based on its direction. The brightest parts of an object become those pointing directly at the light. This is done using the vertex normals to calculate the light intensity for each pixel.

The intensity of the diffuse light for each pixel is calculated and then used to add additional colour to the ambient colour present in the scene.

$$light_i = pixel_n \cdot light_d \cdot -1 \quad (8)$$

$light_i$ is intensity, $pixel_n$ is the the normals, $light_d$ is the direction which must be inverted.

$$colour = (light_a + light_c \cdot light_i) \cdot pixel_c \cdot texture_c \quad (9)$$

$light_a$ is ambient light colour, $light_c$ is the directional light colour, $pixel_c$ is the initial pixel colour based on the vertices colour, and $texture_c$ is the colour of a texture after sampling.

3.2 Framework

3.2.1 Shader Class

This project uses a shader class to handle the vertex and pixel shaders. Shaders are compiled from a file. The shaders are then created and the input layout is setup with per-vertex and per-instance data.

The vertex and pixel shader constant buffers are then created with the D3D11_CPU_ACCESS_WRITE access flag. The light buffer is padded to 16-byte alignment. The constant buffer for the vertex shader contains the view and projection matrices while the constant buffer for the pixel shader contains the directional light colour and direction.

A sampler state is created for the pixel shader so it can sample pixel colours from textures.

The constant buffers are updated each frame and a ID3D11ShaderResourceView for the texture is passed to the pixel shader on each draw call.

3.2.2 Models

A model object is created to load appropriate vertex information and texture file. Instancing is used for each model by reusing common data, allowing for batched draw calls. The required vertex and instance data is defined via structs. Vertex data is reused for every instance of a model.

```
1 struct Vertex
2 {
3   dx::XMFLOAT3 position;
4   dx::XMFLOAT2 texture;
5   dx::XMFLOAT3 normal;
6 };
```

```
1 struct Instance
2 {
3   dx::XMFLOAT3 position;
4   dx::XMFLOAT3 rotation;
5   dx::XMFLOAT3 scale;
6   dx::XMFLOAT4 colour;
7 };
```

An instance buffer is created with the model with a small initial size and the D3D11_CPU_ACCESS_WRITE access flag.

When an instance is added, removed, or changed, the instance buffer needs to be updated. The instance buffer is updated every frame prior to rendering by mapping with D3D11_MAP_WRITE_DISCARD and is grown when required by recreating it.

3.2.3 Textures

A texture object is created that contains a ID3D11ShaderResourceView and ID3D11Texture2D. The texture is loaded from a TGA file. This is composed of a header defining the width, height, and bits per pixels, and a series of bytes representing each pixels colour. The TGA file is used to define the width & height of a texture description object and create the texture resource with the TGA pixel data.

3.2.4 Ray-triangle intersection

Ray-triangle intersection is used to get the intersection position of mouse clicks so that it can be used to command units to move to the closest cell of that intersection. It is also used to vertically sample the terrain to create A* nodes at the intersection points, creating a grid.

Ray-triangle intersection algorithm using Barycentric coordinates is used.

Two edges of the triangle are calculated and then the normals are calculated by multiplying the edges together.

$$normal = \begin{pmatrix} e1_y \cdot e2_z \\ e1_z \cdot e2_x \\ e1_x \cdot e2_y \end{pmatrix} - \begin{pmatrix} e1_z \cdot e2_y \\ e1_x \cdot e2_z \\ e1_y \cdot e2_x \end{pmatrix} \tag{10}$$

the magnitude of the normals is then calculated to normalize the normals

$$\frac{normal}{\sqrt{n_x^2 + n_y^2 + n_z^2}} \tag{11}$$

The distance from the origin to the triangle is found and the dot of the normals and the ray is done to get the denominator. the numerator is the inverse of the normals dot ray origin. The intersection is then the numerator over the denominator and the intersection vector is $start + (dir * t)$. Normals are calculated for every edge and the determinant is calculated and checked to see if it is inside our outside the triangle. If the intersection for all edges is inside the triangle then the intersection is valid.

3.2.5 Camera

The camera controls the view matrix. The view matrix is created for a left-hand coordinate system by using the camera position, the up vector, and a focal point.

```
1 dx::XMMATRIX rotation_matrix = dx::
    XMMatrixRotationRollPitchYaw(
    pitch, yaw, roll);
2 lookat_vector = dx::
    XMVector3TransformCoord(
    lookat_vector, rotation_matrix);
3 up_vector = dx::
    XMVector3TransformCoord(up_vector
    , rotation_matrix);
4 focalpoint = dx::XMVectorAdd(
    position_vector, lookat_vector);
5 view_matrix = dx::XMMatrixLookAtLH(
    position_vector, focalpoint,
    up_vector);
```

3.3 AI

3.3.1 RTS

3.3.2 World-space mouse projection

Movement commands for AI units require projecting mouse coordinates in to world-space from screen-space. This is done by unprojecting the mouse coordinates using the projection and view matrices with the viewport information.

```
1 near_pos_vec = dx::
    XMVector3Unproject(
2 near_pos_vec,
3 0, 0,
4 width, height,
5 0.0f, 1.0f,
6 projection_matrix,
7 view_matrix,
8 world_matrix);
```

3.3.3 Pathfinding

A* can find the shortest distance between two nodes using a heuristic function to determine the estimated cost, known as H to move between nodes, and the actual cost to move between nodes, known as G. The total cost of a node is $F = G + H$. If the heuristic distance is ever larger than the true distance then the A* search is not guaranteed to find the shortest path.

A 2D grid of nodes with 3D positions is created by sampling every triangle in the terrain vertically using ray-triangle intersection, with each intersection being spaced out equally. Nodes below or above a certain height are unwalkable due to being water or mountainous terrain.

A* works by maintaining an open and closed list of nodes. Nodes in an open list are being checked, while nodes in a closed list will form the final path from the starting node to the goal node. The starting node forms the beginning of the open list. The neighbouring nodes of the starting node are added to the open list and the starting node is then moved to the closed list.

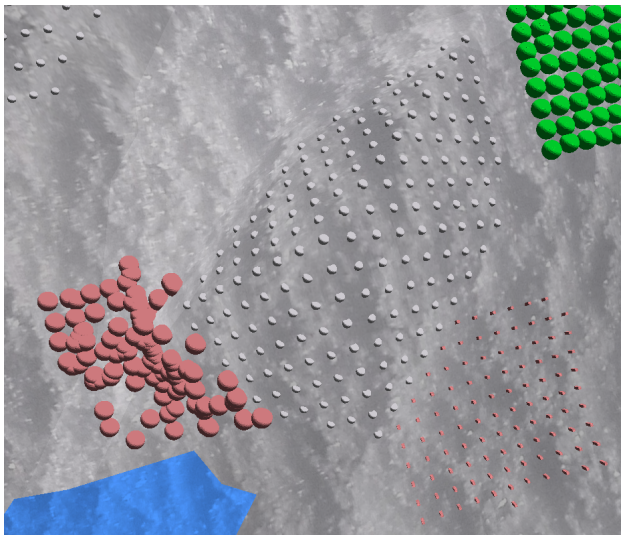

Figure 2: *Using the modified octile distance heuristic, RTS units path around the base of a mountain instead of over it.*

A loop is then performed until the goal node is part of the closed list thus forming our path, or we run out of open cells, in which case no path was found.

On each iteration a node in the open list with the lowest F cost is selected and moved to the closed list, making it the current node.

The neighbouring nodes of that node are added to the open list, ignoring duplicates and any nodes in the closed list. The costs for each of the neighbouring cells are recalculated based on the current cells G cost plus the H cost from the current cell to its neighbours. If this recalculated cost is better than the current neighbouring cell G cost then the current node is made a parent of its neighbours. Any cell not in the open list is added with these calculated costs.

Once a path is found a list of nodes is constructed, starting from the goal node and working backwards by grabbing the parents of each node. The list is then reversed, resulting in the final path.

3.3.4 Heuristic

A modified octile distance heuristic is used for calculating the H cost of each node.An orthogonal movement cost of 100 referred to as D is used with a diagonal movement cost of 140 which is an estimate of $\sqrt{(2D)}$, referred to as $D2$. The heuristic works by calculating the steps distance when not using diagonals and subtracting the steps that were saved by using them. The number of diagonal steps is $min(dist_x, dist_y)$ with every step having a cost of $D2$ yet saving $2D$ orthogonal steps.

The octile distance heuristic is calculated by taking the absolute difference between grid positions of the nodes.

$$dist = |\begin{pmatrix} node_x - goal_x \\ node_y - goal_y \end{pmatrix}| \tag{12}$$

and running it through the following equation

$$h = D(dist_x + dist_y) + (D2 - 2D) \cdot min(dist_x, dist_y) \tag{13}$$

The only difference between this and the Chebyshev distance heuristic is that the diagonal movement cost is $\sqrt{(2D)}$ instead of $2D$. Octile is better when movement is performed in terms of real space and not a grid due to accurate diagonal distances.

A modified version of the octile distance heuristic is used to add a vertical movement penalty, incentivizing the creation of paths around mountains and valleys. This height penalty is calculated using the following equation, with $node_a$ and $goal_a$ being the real-world height values of the node.

$$penalty = 2D(|node_a - goal_a|) \tag{14}$$

$penalty$ is then added to h to form our final heuristic.

3.3.5 Squads

Squads consist of several units. When a movement command is given a flood fill is performed around the goal node. To perform the flood fill neighbouring nodes are gathered until enough unique and walkable nodes have been found, one for every unit in a squad. These form a list of valid nodes. The list of valid nodes may contain more nodes than the number of units so it's sorted based on the heuristic function from itself to the goal node, ensuring the closest neighbouring nodes are used. This maintains the formation at the end of the path. Each unit then finds a path from their position to one of the nearby goal nodes, starting from the closest goal node in the list.

4 Evaluation

4.1 Heuristic

The modified octile distance heuristic works well in ensuring that the path taken by units is realistic. As can be seen in Figure 2 the units of the squad are pathing around the base of the mountain. There are issues with this home-grown solution in that over long distances the heuristic becomes innacurate. This issue is rarely seen within the prototype and may be caused by the estimated distance of the heuristic being longer than that of the real distance, causing A* to lose its property of finding the shortest path.

4.2 Squad Splitting

The technique of using flood fill on the goal node to find desirable goal positions for every unit of a squad

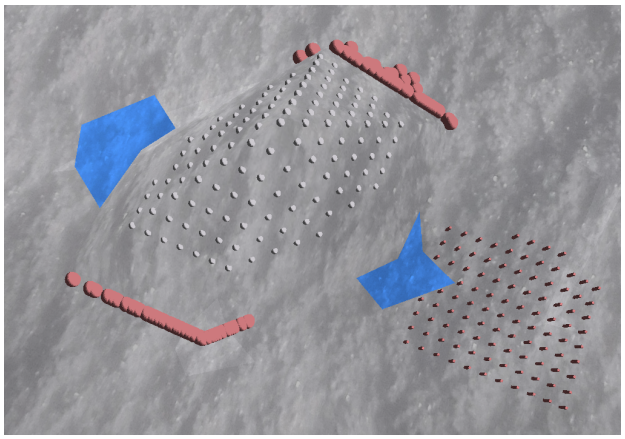

Figure 3: *The members of a squad split around obstacles due to their individually shorter paths*

worked well to ensure the end goals were correct but does not impact the cost of generating each path so as to ensure that units in the same squad stay together, even if they get there a little slower. This can be seen in Figure 3 where the squad splits in two to path around a mountain in the quickest way possible.

5 Conclusion

The framework could be further improved by adding additional lighting types, such as point lights, and additional light shading, such as phong. Shadows could also be added. Additional texture loading options besides TGA would be useful and transparent objects are not implemented.

The RTS AI requires a few improvements. A better heuristic that allows A* to find the shortest path is required and in a functioning RTS game it would not be desirable to have units in a squad split, so that issues must be fixed as well. Additional work should be done to ensure that a squad maintains their formation while traveling when possible and to avoid overlapping eachother in the same positions.

The task was successfully accomplished with an effective RTS AI and framework.

Bibliography

Cui, Xiao and Hao Shi (2011). “A*-based pathfinding in modern computer games”. In: *International Journal of Computer Science and Network Security* 11.1, pp. 125–130.

Kennedy, John and Michael Sartran (2018). *How to Use Direct3D 11 - Windows applications*. URL: `https://docs.microsoft.com/en-us/windows/desktop/direct3d11/how-to-use-direct3d-11`.

Plunkett, Luke (2009). URL: `https://kotaku.com/and-the-first-directx11-game-is-5371466`.